Path-Tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs

RUOMING JIN, Kent State University
NING RUAN, Kent State University
YANG XIANG, The Ohio State University
HAIXUN WANG, Microsoft Research Asia

Reachability query is one of the fundamental queries in graph database. The main idea behind answering reachability queries is to assign vertices with certain labels such that the reachability between any two vertices can be determined by the labeling information. Though several approaches have been proposed for building these reachability labels, it remains open issues on how to handle increasingly large number of vertices in real world graphs, and how to find the best tradeoff among the labeling size, the query answering time, and the construction time. In this paper, we introduce a novel graph structure, referred to as *path-tree*, to help labeling very large graphs. The path-tree cover is a spanning subgraph of G in a tree shape. We show path-tree can be generalized to chain-tree which theoretically can has smaller labeling cost. On top of path-tree and chain-tree index, we also introduce a new compression scheme which groups vertices with similar labels together to further reduce the labeling size. In addition, we also propose an efficient incremental update algorithm for dynamic index maintenance. Finally, we demonstrate both analytically and empirically the effectiveness and efficiency of our new approaches.

Categories and Subject Descriptors: H.2.8 [**Database management**]: Database Applications—*graph indexing and querying*

General Terms: Performance

Additional Key Words and Phrases: Graph indexing, reachability queries, transitive closure, path-tree cover, maximal directed spanning tree

ACM Reference Format:
Jin, R., Ruan, N., Xiang, Y., and Wang, H. 2011. Path-Tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs ACM Trans. Datab. Syst. 1, 1, Article 1 (January 2011), 52 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Ubiquitous graph data coupled with advances in graph analyzing techniques are pushing the database community to devote more attention to graph databases. Efficiently managing and answering queries against very large graphs is becoming an increasingly important research topic driven by many emerging real world applications: Se-

Results of this paper were partially presented at the SIGMOD'08 conference [Jin et al. 2008]. R. Jin and N. Ruan were partially supported by the National Science Foundation, under grant IIS-0953950. Y. Xiang was supported in part by the National Science Foundation under Grant #1019343 to the Computing Research Association for the CIFellows Project.
Author's addresses: R. Jin and N. Ruan, Computer Science Department, Kent State University, Email: {jin,nruan}@cs.kent.edu; Y. Xiang, Department of Biomedical Informatics, The Ohio State University, Email: yxiang@bmi.osu.edu; H. Wang, Microsoft Research Asia, Email: haixunw@microsoft.com.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

mantic Web (XML/RDF/OWL), social network analysis, and bioinformatics, to name a few.

Among them, graph reachability queries have attracted a lot of research attention. Given two vertices u and v in a directed graph, a reachability query asks if there is a path from u to v. Graph reachability is one of the most common queries in a graph database. In many applications where graphs are used as the basic data structure (e.g., XML data management, social network analysis, ontology query processing), reachability is also one of the fundamental operations. Thus, efficient processing of reachability queries is critical in graph databases.

1.1. Applications

Reachability queries are very important for many XML databases. Typical XML documents are tree structures, where reachability queries simply correspond to ancestor-descendant search ("//"). However, with the widespread use of ID and IDREF attributes, it is more appropriate to represent XML documents as directed graphs. Queries on such data often involve reachability. For instance, in bibliographic data which contains a paper citation network, such as in Citeseer, we may ask if author A is influenced by paper B, which can be represented as a non-standard path expression `//B//A`. Such a path, however, is not constrained by the tree structure, but rather, embodied by IDREF links. A typical way of processing this query is to obtain (possibly through some index on elements) elements A and B and then test if author A is reachable from paper B in the XML graph. Clearly, it is crucial to provide efficient support for reachability testing due to its importance for complex XML queries.

Querying ontologies is becoming increasingly important as many large domain ontologies are being constructed. One of the most well-known ontologies is the gene ontology (GO) [1]. GO can be represented as a directed acyclic graph (DAG) in which nodes represent concepts (vocabulary terms) and edges relationships (*is-a*, *part-of*, etc.). It provides a controlled vocabulary to describe a gene product, e.g., proteins or RNAs, in any organism. For instance, we may query if a certain protein is related to a certain biological process or has a certain molecular function. In the simple case, this can be transformed into a reachability query on two vertices over the GO DAG. As a protein can directly associate with several vertices in the DAG, the entire query process may actually invoke several reachability queries.

Recent advances in system biology have amassed a large amount of graph data, including for example, various kinds of biological networks: gene regulatory, protein-protein interaction, signal transduction, metabolic, etc. As many databases are being designed for such data, biology and bioinformatics are becoming a driving force for advances in graph databases. Here again, reachability is one of the fundamental queries frequently used. For instance, we may ask if one gene is (directly or indirectly) regulated by another gene, or if there is a biological pathway between two proteins.

1.2. Prior Work

In order to tell whether a vertex u can reach another vertex v in a directed graph $G = (V, E)$, we can use two "extreme" approaches. The first approach traverses the graph (by Depth-First Search or Breadth-First Search) trying to find a path between u and v, which takes $O(n + m)$ time, where $n = |V|$ (number of vertices) and $m = |E|$ (number of edges). This is apparently too slow for large graphs. The other approach precomputes the transitive closure of G, i.e., it records the reachability between any pair of vertices in advance. While this approach can answer reachability queries in $O(1)$ time, the computation of transitive closure has complexity of $O(mn)$ [Simon 1988]

[1] http://www.geneontology.org

and the storage cost is $O(n^2)$. Both are unacceptable for large graphs. Existing research has been trying to find good ways to reduce the precomputation time and storage cost with reasonable answering time.

A key idea explored by existing research is to utilize simpler graph structures, such as chains or trees, in the original graph to compute and compress the transitive closure and/or help with reachability answering.

The Chain Decomposition Approach. Chains are the first simple graph structure that has been studied in both graph theory and database literature to improve the efficiency of the transitive closure computation [Simon 1988] and to compress the transitive closure matrix [Jagadish 1990]. The basic idea of chain decomposition is as follows: a DAG is partitioned into several pair-wise disjoint chains (one vertex appears in one and only one chain). Each vertex in the graph is assigned a chain number and its sequence number in the chain. For any vertex v and any chain c, we record at most one vertex u such that u is the smallest vertex (in terms of u's sequence number) on chain c that is reachable from v. To tell if any vertex x reaches any vertex y, we only need to check if x reaches any vertex y' in y's chain and y' has a smaller sequence number than y.

Currently, Simon's algorithm [Simon 1988], which uses chain decomposition to compute the transitive closure, has worst case complexity $O(k \cdot e_{red})$, where k is width (the total number of chains) of the chain decomposition and e_{red} is the number of edges in the transitive reduction of the DAG G (the transitive reduction of G is the smallest subgraph of G which has the same transitive closure as G, $e_{red} \leq e$). Jagadish *et al.* [Jagadish 1990] applied chain decomposition to reduce the size of the transitive closure matrix. They derived the minimal number of chains of G by transforming the problem into an equivalent network flow problem, which can be solved in $O(n^3)$, where n is the number of vertices in DAG G. Several heuristic algorithms have been proposed to reduce the actual index cost of chain decomposition.

Though chain decomposition can help compress the transitive closure, its compression rate is limited by the fact that each node can have no more than one immediate successor. In many applications, even though the graphs are rather sparse, each node can have multiple immediate successors, and the chain decomposition approach considers at most one of them.

The Tree Cover Approach. Instead of using chains, Agrawal *et al.* used a (spanning) tree to "cover" the graph and compress the transitive closure matrix. They showed that the tree cover can beat the best chain decomposition [Agrawal et al. 1989]. The proposed algorithm finds the best tree cover that can maximally compress the transitive closure. The cost of such a procedure, however, is in worst case equivalent to computing the transitive closure.

The tree cover approach is based on interval labeling. Given a tree, we assign each vertex a pair of numbers (an interval). If vertex u can reach vertex v, then the interval of u contains the interval of v. The interval can be obtained by performing a postorder traversal of the tree. Each vertex v is associated with an interval $[i, j]$, where j is the postorder number of vertex v and i is the smallest postorder number among its descendants (each vertex is a descendant of itself).

Assume we have found a tree cover (a spanning tree) of the given DAG G, and vertices of G are indexed by their interval label. Then, for any vertex, it is enough to record the intervals of the nodes that it can reach. In addition, if u reaches the root of a subtree, then it is enough to record the interval of that root vertex as the interval of any other vertex in the subtree is contained by that of the root vertex. To answer

whether u can reach v, we will check if the interval of v is contained by any interval recorded for u.

Other Variants of Tree Covers (Dual-Labeling, Label+SSPI, and GRIPP). Several recent studies tried to address the deficiency of the tree cover approach introduced by Agrawal *et al.* Wang *et al.* [Wang et al. 2006] developed the Dual-Labeling approach which improves the query time and reduces the index size for sparse graphs (the original tree cover approach would cost $O(n)$ and $O(n^2)$, respectively). For very sparse graphs, they claim the number of non-tree edges t is much smaller than n ($t << n$). Their approaches can reduce the index size to $O(n + t^2)$ and achieve constant query answering time. Their major idea is to build a transitive link matrix, which can be thought of as the transitive closure for the non-tree edges. Basically, each non-tree edge is represented as a vertex and a pair of them is linked if the starting of one edge v can be reached by the end of another edge u through the interval index (v is u's descendant in the tree cover). They develop approaches to utilize this matrix to answer the reachability query with constant time. In addition, the tree generated in dual-labeling is different from the optimal tree cover, as here the goal is to minimize the number of non-tree edges. This is essentially equivalent to the transitive reduction computation which has proved to be as costly as the transitive closure computation. Thus, their approach (including the transitive reduction) requires an additional $O(nm)$ construction time if non-tree edges should be minimized. Clearly, the major issue of this approach is that it depends heavily on the number of non-tree edges. If $t > n$ or $m_{red} \geq 2n$, this approach will not help with the computation of transitive closure, or compress the index size.

Label+SSPI [Chen et al. 2005] and GRIPP [Trißl and Leser 2007] aim to minimize the index construction time and index size. They achieve $O(m + n)$ index construction time and $O(m + n)$ index size. However, this is at the sacrifice of the query time, which will cost $O(m-n)$. Both algorithms start by extracting a tree cover. Label+SSPI utilizes pre- and post-order labeling for a spanning tree and an additional data structure for storing non-tree edges. GRIPP builds the cover using a depth-first search traversal, and each vertex which has multiple incoming edges will be duplicated accordingly in the tree cover. In some sense, their non-tree edges are recorded as non-tree vertex instances in the tree cover. To answer a query, both of them will deploy an online search over the index to see if u can reach v. GRIPP has developed a couple of heuristics which utilize the interval property to speed up the search process.

2-HOP Labeling. The 2-hop labeling method proposed by Cohen *et al.* [Cohen et al. 2003] represents a quite different approach. Intuitively, it tries to identify a subset of vertices V_s in the graph that best capture the connectivity information of the DAG. Then, for each vertex v in the DAG, it records a list of vertices in V_s that can reach v, denoted as $L_{in}(v)$, and a list of vertices in V_s that v can reach, denoted as $L_{out}(v)$. These two sets record all the necessary information to infer the reachability of any pair of vertices u and v, i.e., if $u \rightarrow v$, then $L_{out}(v) \cap L_{in}(v) \neq \emptyset$, and vice versa. For a given labeling, the index size is $I = \sum_{v \in V} |L_{in}(v)| + |L_{out}(v)|$. They propose an approximate (greedy) algorithm based on set-covering which can produce a 2-hop cover with size no larger than the minimum possible 2-hop cover by a logarithmic factor. The minimum 2-hop cover is conjectured to be $\tilde{O}(nm^{1/2})$. However, in order to find the good 2-hop cover, their original algorithm requires $O(n*f(n)*|T_c|)$ time to compute the transitive closure first (Recall there are n auxiliary undirected bipartite graphs and the ground set to be covered is T_c, the transitive closure), where $f(n)$ is the time to compute the densest subgraph of a graph G with n vertices by the 2-approximation algorithm. In [Cohen et al. 2003], Cohen *et al.* claimed that the algorithm takes linear time, but did not men-

tion explicitly what it is linear to. Our analysis shows that it is linear to the number of edges in the undirected bipartite graph and therefore $O(f(n)) = O(n^2)$.

Recently, several approaches have been proposed to reduce the construction time of 2-hop. Schenkel *et al.* proposed the HOPI algorithm, which applies a divide-and-conquer strategy to compute 2-hop labeling [Schenkel et al. 2004]. Their algorithm is heuristic and does not reduce the worst cast complexity of the construction time. Cheng *et al.* [Cheng et al. 2006] proposed a geometric-based algorithm to produce a 2-hop labeling. Their algorithm does not require the computation of transitive closure, but does not produce the approximation bound of the labeling size which is produced by Cohen's approach.

3-HOP Labeling. The 3-hop reachability labeling proposed by Jin *et al.* [Jin et al. 2009] tries to simulate the highway system of the transportation network. To reach a destination from a starting point, one simply needs to get on an appropriate highway and get off at the right exit to the destination. The authors study how to use chain structure to serve as the highway. Given this, the three hops are 1) the first hop from the starting vertex to the entry point of some chain, 2) the second hop from the entry point in the chain to the exit point of the chain, and finally 3)the third hop from the exit point of the chain to the destination vertex. Thus, 3-hop labeling generalizes 2-hop by replacing those intermediate vertices V_s with chain structures.

Using the 3-hop scheme, the authors first demonstrate that the chain decomposition naturally introduces the set of *contour points* $Con(G)$, which corresponds to the essential reachability transition between any two chains. The set of contour points can uniquely recover the full transitive closure and can even be used to directly answer the reachability queries. Specifically, each contour point is a vertex pair (u, v), where u is referred to as an *out-anchor* vertex and v is an *in-anchor* vertex. Then, a 3-hop reachability labeling further "factorizes" those contour points by assigning each out-anchor vertex u of $Con(G)$ a label $L_{out}(u)$ (a set of intermediate entry points), and each in-anchor vertex v a label $L_{in}(v)$ (a set of intermediate exit point). For any $(u, v) \in Con(G)$, there is at least $x \in L_{out}(u)$ and $y \in L_{in}(v)$, such that x and y in the same chain and $x \rightarrow y$. Basically, the two sets record the necessary information to recover $Con(G)$ and further infer any reachability information in the graph. Similar to 2-hop, the 3-hop approach also relies on the greedy set-cover approach to approximate the minimal labeling size, which is denoted as $\sum_u |L_{out}(u)| + \sum_v |L_{in}(v)|$.

The 3-hop is proved to have better compression ratio compared with 2-hop. However, the major issue of 3-hop is its computational cost, which has the worst case complexity $O(kn^2|Con(G)|)$, where k is the number of chains. Though it is faster than 2-hop, it is still too high to be scalable. It remains an open issue to scale the set-covered based approaches, like 2-hop and 3-hop, without comprising the approximation bound.

GRAIL. GRAIL is the latest scalable reachability indexing scheme introduced by Yildirim *et al.* [Yildirim et al. 2010]. Basically, each vertex u in the DAG is assigned with multiple interval labels L_u which can help quickly determine the non-reachability between two vertices. These labels are generated by performing a constant number (d) of *random* depth-first traversals, i.e., the visiting order of the neighbors of each vertex is randomized in each traversal. Each traversal will produce one interval for every vertex in the graph. Especially, such interval labeling has the property that if $L_v \nsubseteq L_u$, then vertex u cannot reach vertex v. However, when $L_v \subseteq L_u$, we cannot determine whether u can reach v. Thus, $L_v \subseteq L_u$ is a necessary but insufficient condition for determining the reachability between u and v. In [Yildirim et al. 2010], Yildirim *et al.* utilize this labeling in the depth-first search to prune the search space. The advantage of this approach is that its index can be constructed very fast ($O(d(n+m))$) and its index size is only determined by the number of intervals (d) and the number of vertices in

Table I. Complexity comparison

	Query time	Construction time	Index size
Transitive Closure	$O(1)$	$O(nm)$[1]	$O(n^2)$
Opt. Chain Cover[2]	$O(k)$	$O(nm)$	$O(nk)$
Opt. Tree Cover [3]	$O(n)$	$O(nm)$	$O(n^2)$
2-Hop[4]	$\tilde{O}(m^{1/2})$	$O(n^3\|T_c\|)$	$\tilde{O}(nm^{1/2})$
Dual Labeling[5]	$O(1)$	$O(n+m+t^3)$	$O(n+t^2)$
Labeling+SSPI	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRIPP	$O(m-n)$	$O(n+m)$	$O(n+m)$
3-Hop	$O(\log n+k)$ or $O(n)$	$O(kn^2\|Con(G)\|)$	$O(nk)$
GRAIL [6]	$O(d)$ to $O(n+m)$	$O(d(n+m))$	$O(dn)$
Path-tree[7]	$O(\log^2 k)$	$O(mk)$	$O(nk)$

the graph. However, in the worst case, this approach can be downgraded to DFS which takes $O(n+m)$ in query processing.

An early version of the Path-Tree approach. We have developed an early version of the path-tree algorithm [Jin et al. 2008] for graph reachability. In this paper, we have not only completed the theory and the algorithms introduced in the early version, but also significantly extends the path-tree approach in three key directions: 1) we generalize path-tree to chain-tree and prove the optimality of chain-tree indexing (Section 4); 2) we introduce a new compression scheme by further reducing the index size of the path-tree and chain-tree without sacrificing the query processing time (Section 5). 3) we also provide proofs of all lemmas and theorems and improve the index construction time from $O(mk^2)$ to $O(mk)$ (Section 2.5). 4) we perform a very thorough empirical-study between two versions of path-trees and their new compression improvements, with the state-of-art reachability indexing schemes including the tree-cover [Agrawal et al. 1989], 2-hop [Cohen et al. 2003], 3-hop [Jin et al. 2009], and GRAIL [Yildirim et al. 2010]. In addition, in the appendix, we also provide efficient incremental update approaches.

Beyond Simple Reachability. Several recent studies have gone beyond simple reachability query in the direct graph to consider additional constraints in different applications. Bouros *et al.* [Bouros et al. 2009] studied how to evaluate reachability queries over a set of constantly evolving paths. The examples of such path-collections include the set of biological pathways and popular touristic route archive. Though we can aggregate the path to generate the underlying directed graph and then answer reachability query on this graph, the authors argue this is not efficient due to the dynamic nature of path collection. They have proposed an $\mathcal{H}-Index$ to capture the path-path relationship, and utilized it to facilitate the search process. Jin *et al.* [Jin et al. 2010] study the reachability problem in edge-labeled graphs, where each edge is associated with a label that denotes the relationship between the two vertices connected by the edge. Specifically, they introduce the *label-constraint reachability query*: Can vertex u reach vertex v through a path whose edge labels are constrained by a set of labels? They generalize the transitive closure in the labeled graph and propose a novel indexing framework based on maximal directed spanning tree and sampling techniques to maximally compress the labeled transitive closure.

1.3. Our Contribution

In Table I we show the indexing and querying complexity of different reachability approaches. Throughout the above comparison and several existing studies [Trißl and Leser 2007; Wang et al. 2006; Schenkel et al. 2004], we can see that even though the 2-hop approach is theoretically appealing, it is rather difficult to apply it on very large graphs due to its computational cost. At the same time, as most large graphs are rather sparse, the tree-based approach seems to provide a good starting point to compress the transitive closure and to answer reachability queries. Most recent studies tried to improve different aspects of the tree-based approach [Agrawal et al. 1989; Wang et al. 2006; Chen et al. 2005; Trißl and Leser 2007]. Since we can effectively transform any directed graph into a DAG by contracting strongly connected components into vertices and utilizing the DAG to answer the reachability query, we will only focus on DAG for the rest of the paper.

Our study is motivated by a several challenging issues that tree cover based approaches do not adequately address. First, the computational cost of finding a good tree cover can be expensive. For instance, it costs $O(mn)$ to extract a tree cover with Agrawal's optimal tree cover [Agrawal et al. 1989] and Wang's Dual-labeling tree [Wang et al. 2006]. Second, the tree cover cannot represent some common types of DAGs, for instance, the Grid type of DAG [Schenkel et al. 2004], where each vertex in the graph links to its right and upper corners. For a $k \times k$ grid, the tree cover can maximally cover half of the edges and the compressed transitive closure is almost as big as the original one. We believe the difficulty here is that the strict tree structures are too limited to express many different types of DAGs even when they are very sparse. From another perspective, most of the existing methods which utilize the tree cover are greatly affected by how many edges are left uncovered.

Driven by these challenges, in this paper, we propose a novel graph structure, referred to as *path-tree*, to cover a DAG. It creates a tree structure where each node in the tree represents a path in the original graph. Given that many real world graphs are very sparse, e.g., the number of edges is no more than 2 times of the number of vertices, the path-tree provides us a better way to cover the DAG compared with the tree cover. In addition, to answer a reachability query, we develop a labeling scheme where each label has only 3 elements in the path-tree. We show that a good path-tree cover can be constructed in $O(m + n \log n)$ time and the index can be constructed in $O(mk)$ time. Theoretically, we prove that the path-tree cover can always perform the compression of transitive closure better than or equal to the optimal tree cover approaches and chain decomposition approaches.

Furthermore, we study the following key aspects of path-tree indexing: 1) we generalize the path-tree to the chain-tree, which theoretically can produce better indexing size than the path-tree; and 2) inspired by the general graph compression and summarization methods [Adler and Mitzenmacher 2002; Raghavan and Garcia-Molina 2003; Navlakha et al. 2008], we employ a kmeans-type algorithm to group vertices with similar reachability together and utilize the common reachability to further reduce their

[1] m is the number of edges and $O(n^3)$ if using Floyd-Warshall algorithm [Cormen et al. 2001]
[2] k is the width of chain decomposition; Query time can be improved to $O(\log k)$ (assuming binary search) and construction time becomes $O(mn + n^2 \log n)$, which includes the cost of sorting.
[3] Query time can be improved to $O(\log n)$ and construction time becomes $O(mn + n^2 \log n)$.
[4] The index size is still a conjecture.
[5] It requires an additional $O(nm)$ construction time if the number of non-tree edges should be minimized.
[6] d is the number of intervals assigned to each vertex; query time varies from $O(d)$ (non-reachability can be quickly determined using intervals) to $O(n + m)$ (worst case complexity)
[7] For PTree-1, the path-tree is built on optimal tree cover, which takes $O(mn)$ construction time.

index size. Particularly, we propose a fast *sliding-window* method which takes advantage of topological sorting to find a good initial grouping for the kmeans compression algorithm. 3) we perform a thorough experimental evaluation on both real and synthetic datasets. Our results show that the path-tree indexing provides the fastest query processing time on the existing real benchmark graphs and on large sparse graphs. It is also very easy to build and has the comparable index size with respect to the state-of-art indexing methods including tree-cover, 2-hop, and 3-hop. In addition, we provide efficient incremental update algorithms to deal with edge/node insertion and deletion in the graph.

The rest of the paper is organized as follows. In Section 2, we introduce the path-tree concept and an algorithm to construct a path-tree from the DAG. In Section 3, we investigate several optimality questions related to path-tree cover. In Section 4, we introduce the concept of generalizing path-tree to chain-tree, and discuss the optimality of chain-tree. In Section 5, we present a new compression scheme to further reduce the index size generated from the path-tree and the chain-tree. In Section 6, we present the experimental results. We conclude in Section 7. In Appendix, we show how to effective perform incremental updates such as addition or deletion of an edge.

2. PATH-TREE COVER FOR REACHABILITY QUERY

We propose to use a novel graph structure, *Path-Tree*, to cover a DAG G. The path-tree cover is a spanning subgraph of G in a tree shape. Under a labeling scheme we devise for the path-tree cover wherein each vertex is labeled with a 3-tuple, we can answer reachability queries on the path-tree cover (not the entire graph) in $O(1)$ time. Then by utilizing the path-tree cover, we show how the full transitive closure of G can be compressed to answer the transitive closure for the entire graph.

We start with the basic notations which will be used throughout the paper. Let $G = (V, E)$ be a directed acyclic graph (DAG), where $V = \{1, 2, \cdots, n\}$ is the vertex set, and $E \subseteq V \times V$ is the edge set. We use (v, w) to denote the edge from vertex v to vertex w, and we use $(v_0, v_1, \cdots, v_p)$ to denote a *path* from vertex v_0 to vertex v_p, where (v_i, v_{i+1}) is an edge ($0 \leq i \leq p-1$). Because G is acyclic, all vertices in a path must be distinct. We say vertex v is reachable from vertex u (denoted as $u \rightarrow v$) if there is a path starting from u and ending at v.

For a vertex v, we refer to all edges that start from v as *outgoing edges* of v, and all edges ending at v as *incoming edges* of v. The *predecessor set* of vertex v, denoted as $S(v)$, is the set of all vertices that can reach v, and the *successor set* of vertex v, denoted as $R(v)$, is the set of all vertices that v can reach. The successor set of v is also called the *transitive closure* of v. The transitive closure of DAG G is the directed graph where there is a direct edge from each vertex v to any vertex in its successor set.

In addition, we say $G_s = (V_s, E_s)$ is a *subgraph* of $G = (V, E)$ if $V_s \subseteq V$ and $E_s \subseteq E \cap (V_s \times V_s)$. We denote G_s as a *spanning subgraph* of G if it covers all the vertices of G, i.e., $V_s = V$. A *tree* T is a special DAG where each vertex has only one incoming edge (except for the root vertex, which does not have any incoming edge). A *forest* (or *branching*) is a union of multiple trees. A forest can be converted into a tree by simply adding a virtual vertex with edges to the roots of each individual tree. To simplify our discussion, we will use trees to refer both trees and forests.

In this paper, we introduce a novel graph structure called *path-tree cover* (or simply *path-tree*). A path-tree cover for G, denoted as $G[T] = (V, E', T)$, is a spanning subgraph of G and has a tree-like shape which is described by tree $T = (V_T, E_T)$: Each vertex v of G is uniquely mapped to a single vertex in T, denoted as $f(v) \in V_T$, and each edge (u, v) in E' is uniquely mapped to either a single edge in T, $(f(u), f(v)) \in E_T$, or a single vertex in T.

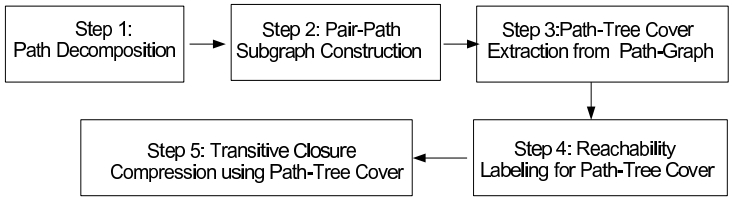


Fig. 1. Overview of Path-Tree Indexing Construction

Figure 1 provides an overview of path-tree indexing construction which contains five key steps. In the following sections, we describe each step in details. Section 2.1 describes how to partition a DAG into paths (step 1). Using this partitioning, we define the *pair-path subgraph* of G and reveal a nice structure of this subgraph (Section 2.2, step 2). We then discuss how to extract a good *path-tree cover* from G (Section 2.3, step 3). We present the labeling schema for the path-tree cover in Section 2.4 (step 4). Finally, we show how the path-tree cover can be applied to compress the transitive closure of G in Section 2.5 (step 5).

2.1. Step 1: Path-Decomposition of DAG

Let P_1, P_2 be two paths of G. We use $P_1 \cap P_2$ to denote the set of vertices that appear in both paths, and we use $P_1 \cup P_2$ to denote the set of vertices that appear in at least one of the two paths. We define graph partitions based on the above terminology.

DEFINITION 1. *Let $G = (V, E)$ be a DAG. We say a partition $P_1, \cdots, P_k$ of V is a path-decomposition of G if and only if $P_1, \cdots, P_k$ are paths of G, and $P_1 \cup \cdots \cup P_k = V$, and $P_i \cap P_j = \emptyset$ for any $i \neq j$. We also refer to k as the width of the decomposition.*

As an example, Figure 2(b) represents a partition of graph G in Figure 2(a). The path decomposition contains four paths $P_1 = \{1, 3, 6, 13, 14, 15\}$, $P_2 = \{2, 4, 7, 10, 11\}$, $P_3 = \{5, 8\}$ and $P_4 = \{9, 12\}$.

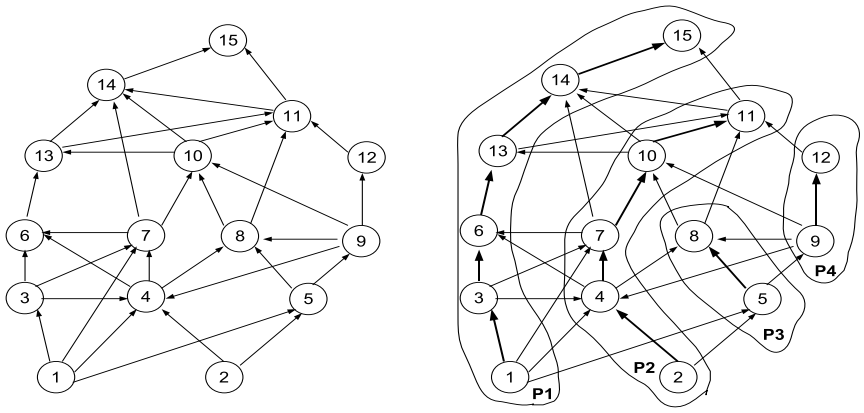
Based on the partition, we can identify each vertex v by a pair of IDs: (pid, sid), where pid is the ID of the path vertex v belongs to, and sid is v's relative order on that path. For instance, vertex 3 in G shown in Figure 2(b) is identified by (1, 2). For two vertices u and v in path P_i, we use $u \preceq v$ to denote u precedes v (or $u = v$) in path P_i:

$$u \preceq v \Longleftrightarrow u.sid \leq v.sid \text{ and } u, v \in P_i$$

NOTE: A simple path-decomposition algorithm is given by [Simon 1988]. It can be described briefly as follows: first, we perform a topological sort of the DAG. Then, we extract paths from the DAG as follows. We find v, the smallest vertex (in the ascending order of the topological sort) in the graph and add it to the path. We then find v', such that v' is the smallest vertex in the graph and there is an edge from v to v'. In other words, we repeatedly add the smallest nodes to the latest extracted vertex until the path could not be extended (the vertex added last has no out-going edges). Then, we remove the entire path (including the edges connecting to it) from the DAG and extract another path. The decomposition is complete when the DAG is empty.

2.2. Step 2: Pair-Path Subgraph and Minimal Equivalent Edge Set

Let us consider the relationships between two paths. We use $P_i \rightarrow P_j$ to denote the pair-path subgraph of G consisting of i) path P_i, ii) path P_j, and iii) $E_{P_i \rightarrow P_j}$, which is the set of edges from vertices on path P_i to vertices on path P_j. For instance, $E_{P_1 \rightarrow P_2} = \{(1, 4), (1, 7), (3, 4), (3, 7), (13, 11)\}$ is the set of edges from vertices in P_1 to vertices in P_2. We say subgraph $P_i \rightarrow P_j$ is *connected* if $E_{P_i \rightarrow P_j}$ is not empty.



(a) (b)

Fig. 2. Path-Decomposition for a DAG

Given a vertex u in path P_i, we want to find all vertices in path P_j that are reachable from u (through paths in subgraph $P_i \rightarrow P_j$ only). It turns out that we only need to know one vertex – the smallest (with regard to sequence id) vertex on path P_j reachable from u. We denote its sid as $r_j(u)$.

$$r_j(u) = min\{v.sid | u \rightarrow v \text{ and } v.pid = j\}$$

Clearly, for any vertex $v' \in P_j$,

$$u \rightarrow v' \Longleftrightarrow v'.sid \geq r_j(u)$$

Certain edges in $E_{P_i \rightarrow P_j}$ can be removed without changing the reachability between any two vertices in subgraph $P_i \rightarrow P_j$. This is characterized by the following definition.

DEFINITION 2. *A set of edges $E^R_{P_i \rightarrow P_j} \subseteq E_{P_i \rightarrow P_j}$ is called the* minimal equivalent edge set *of $E_{P_i \rightarrow P_j}$ if removing any edge from $E^R_{P_i \rightarrow P_j}$ changes the reachability of vertices in $P_i \rightarrow P_j$.*

As shown in Figure 3(a), $\{(3,4),(13,11)\}$ is the minimal equivalent edge set for subgraph $P_1 \rightarrow P_2$. In Figure 3(b), $\{(7,6),(10,13),(11,14)\}$ is the minimal equivalent edge set of $E_{P_2 \rightarrow P_1}$=$\{(4,6),(7,6),(7,14),(10,13),(10,14),(11,14),(11,15)\}$. In Figure 3, edges belonging to the minimal equivalent edge set for subgraphs $P_i \rightarrow P_j$ in G are marked in bold.

In the following, we introduce a property of the minimal equivalent edge set that is important to our reachability algorithm.

DEFINITION 3. *Let (u,v) and (w,z) be two edges in $E_{P_i \rightarrow P_j}$, where $u,w \in P_i$ and $v,z \in P_j$. We say the two edges are* crossing *if $u \preceq w$ (i.e., $u.sid \leq w.sid$) and $v \succeq z$ (i.e., $v.sid \leq z.sid$). Given a set of edges, if no two edges in the set are crossing, then we say they are* parallel.

To understand the above definition of crossing, let us see an example in Figure 3(a). Edge $(1,7)$ and edge $(3,4)$ are crossing in $E_{P_i \rightarrow P_j}$ because $1 \preceq 3$ and $7 \succeq 4$. Now we have the following important lemma:

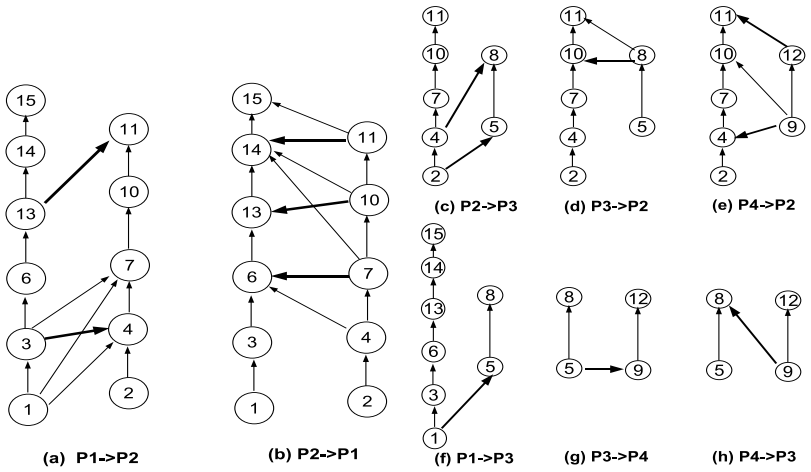


Fig. 3. Path-Relationship of a DAG

LEMMA 1. *No two edges in any minimal equivalent edge set of $E_{P_i \to P_j}$ are crossing, or equivalently, edges in $E^R_{P_i \to P_j}$ are parallel.*

Proof: This can easily be proved by contradiction. Suppose (u, v) and (w, z) are crossing in $E^R_{P_i \to P_j}$. Without loss of generality, let us assume $u \preceq w (u \to w)$ and $v \succeq z (v \leftarrow z)$. Thus, we have $u \to w \to z \to v$. Therefore (u, v) is simply a short cut of $u \to v$, and dropping (u, v) will not affect the reachability for $P_i \to P_j$ as it can still be inferred through edge (w, z). This contradicts the assumption that (u, v) belongs to the minimal equivalent edge set $E^R_{P_i \to P_j}$. Therefore, we prove that edges in $E^R_{P_i \to P_j}$ are parallel. □

After extra edges in $E_{P_i \to P_j}$ are removed, the subgraph $P_i \to P_j$ becomes a simple grid-like planar graph where each node has at most 2 outgoing edges and at most 2 incoming edges. This nice structure, as we will show later, allows us to map its vertices to a two-dimensional space and enables answering reachability queries in constant time.

LEMMA 2. *The minimal equivalent edge set of $E_{P_i \to P_j}$ is unique for subgraph $P_i \to P_j$.*

Proof: We can prove this by contradiction. Assuming the lemma is not true, then there are two different minimal equivalent edge sets of $E_{P_i \to P_j}$, which we call $E^R_{P_i \to P_j}$ and $E^{R'}_{P_i \to P_j}$, having the same reachability information of subgraph $P_i \to P_j$. We sort edges in each set from low to high, using vertex sid in P_i and vertex sid in P_j as primary and secondary keys, respectively. We compare edges in these two sets in sorted order. Assuming $(u, v) \in E^R_{P_i \to P_j}$ and $(u', v') \in E^{R'}_{P_i \to P_j}$ are the first pair of different edges such that $u \neq u'$ or $v \neq v'$, it's easy to get a contradiction that either $E^R_{P_i \to P_j}$ and $E^{R'}_{P_i \to P_j}$ have different reachability information, or one of the sets is not a minimal equivalent edge set. Therefore, the minimal equivalent edge set of $E_{P_i \to P_j}$ is unique. □

A simple algorithm that extracts the minimal equivalent edge set of $E_{P_i \to P_j}$ is sketched in Algorithm 1. We order all the edges from P_i to P_j ($E_{P_i \to P_j}$) by their end vertex in P_j. Let v' be the first vertex in P_j which is reachable from P_i. Let u' be the last vertex in P_i that can reach v'. Then, we add (u', v') into $E^R_{P_i \to P_j}$ and remove all the edges in $E_{P_i \to P_j}$ which start from a vertex in P_i which precedes u' (or equivalently,

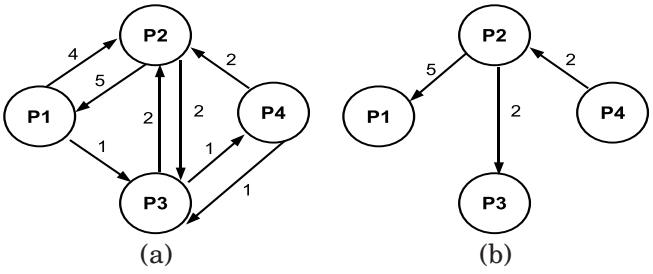


Fig. 4. (a) Weighted Directed Path-Graph & (b) maximum Directed Spanning Tree

which cross edge (u', v')). We repeat this procedure until the edge set $E_{P_i \to P_j}$ becomes empty.

Algorithm 1 MinimalEquivalentEdgeSet(P_i,P_j,$E_{P_i \to P_j}$)

1: $E^R_{P_i \to P_j} = \emptyset$
2: **while** $E_{P_i \to P_j} \neq \emptyset$ **do**
3: $v' \to min(\{v|(u,v) \in E_{P_i \to P_j}\})$ {the first vertex in P_j that P_i can reach}
4: $u' \leftarrow max(\{u|(u,v') \in E_{P_i \to P_j}\})$
5: $E^R_{P_i \to P_j} \leftarrow E^R_{P_i \to P_j} \cup \{(u', v')\}$
6: $E_{P_i \to P_j} \leftarrow E_{P_i \to P_j} \backslash \{(u,v) \in E_{P_i \to P_j} | u \preceq u'\}$ {Remove all edges which cross (u', v')}
7: **end while**
8: return $E^R_{P_i \to P_j}$

2.3. Step 3: Path-Graph and its Spanning Tree ($\mathcal{S}P$-tree)

We create a directed *path-graph* for DAG G as follows. Each vertex i in the path-graph correponds to a path P_i in G. If path P_i connects to P_j in G, we create an edge (i, j) in the path graph. Let T be a directed spanning tree (or a forest) of the path-graph. We refer T as the $\mathcal{S}P$-tree of the path-graph. Let $G[T]$ be the subgraph of G that contains i) all the paths of G, and ii) the minimal edge sets, $E^R_{P_i \to P_j}$, for every i, j if edge $(i, j) \in E(T)$. We will show that there is a vector labeling for $G[T]$ which can answer the reachability query for $G[T]$ in constant time. We refer to $G[T]$ as the *path-tree cover* for DAG G.

Just like Agrawal *et al.*'s tree cover [Agrawal et al. 1989], in order to utilize the path-tree cover, we need to "remember" those edges that are not covered by the path-tree. Naturally, we would like to minimize the number of the non-covered edges, which minimizes the index size. Meanwhile, unlike the tree cover, we want to avoid computing the predecessor set (computing the predecessor set of each vertex is equivalent to computing the transitive closure). In the next subsection, we will investigate how to find the optimal path-tree cover if the knowledge of predecessor set is available. Here, we introduce a couple of alternative criteria which can help reduce the index size without such knowledge.

The first criterion is referred to as *MaxEdgeCover*. The main idea is to use the path-tree to cover as many edges in DAG G as possible. Let t be the remaining edges in DAG G (edges not covered by the path-tree). As we will show later in this subsection, t provides an upper-bound for the compression of transitive closure for G, i.e., each

vertex needs to record at most t vertices for answering a reachability query. Given this, we can simply assign $|E_{P_i \to P_j}|$ as the cost for edge (i, j) in the directed path-graph.

The second criterion is referred to as *MinPathIndex*. As we mentioned, each vertex needs to remember at most one vertex on any other path to answer a reachability query. Given two paths P_i, P_j, and their link set $E_{P_i \to P_j}$, we can quickly compute the index cost as follows if $E_{P_i \to P_j}$ does not include the tree-cover. Let u be the last vertex in path P_i that can reach path P_j. Let $P_i[\to u] = \{v | v \in P_i, v \preceq u\}$ be the subsequence of P_i that ends with vertex u. For instance, in our running example, vertex 13 is the last vertex in path P_1 which can reach path P_2, and $P_1[\to 13] = \{1, 3, 6, 13\}$ (Figure 3). We assign a weight $w_{P_i \to P_j}$ to be the size of $P_i[\to u]$. In our example, the weight $w_{P_1 \to P_2} = 4$. Basically, this weight is the labeling cost if we have to materialize the reachability information for path P_i about path P_j. Considering path P_1 and P_2, we only need to record vertex 4 in path P_2 for vertices 1 and 3 in path P_1 and vertex 11 for vertices 6 and 13. Then, we can answer if any vertex in P_2 can be reached from any vertex in P_1. Thus, finding the maximum spanning tree in such a weighted directed path-graph corresponds to minimizing the index size by using path-tree. Figure 4(a) is the weighted directed path-graph using the *MinPathIndex* criteria.

To reduce the index size for the path-tree cover, we would like to extract the maximum directed spanning tree (or forest). As an example, Figure 4(b) is the maximum directed spanning tree extracted from the weighted directed path-graph of Figure 4(a). The Chu-Liu/Edmonds algorithm can be directly applied to this problem [Chu and Liu 1965; Edmonds 1967]. The fast implementation that uses the Fibonacci heap requires $O(m' + k \log k)$ time complexity, where k is the width of path-decomposition and m' is the number of directed edges in the weighted directed path-graph [Gabow et al. 1986]. Clearly, $k \leq n$ and $m' \leq m$, m is the number edges in the original DAG.

2.4. Step 4: Reachability Labeling for Path-Tree Cover

The path-tree is formed after the minimal equivalent edge sets and the maximum directed spanning tree (maximum $\mathcal{SP}$-tree) are established. In this section, we introduce a vector labeling scheme for vertices in the path-tree. The labeling scheme enables us to answer reachability queries in constant time. We use $G[P]$ to denote the path-tree represented in a special linked-list format which prioritizes the path a vertex belongs to:

> $\forall v \in V : linkedlist(v)$ records all the immediate neighbors of v. Let $v \in P_i$. *If v is not the last vertex in path P_i, the first vertex in the linked list is the next vertex of v in the path*

The purpose of defining $G[P]$ will be clear in Algorithm 2.

To help understanding the reachability labeling for path-tree cover, we start with a simple path-path scenario, i.e., path-path relationship itself forms a path, a special case of tree.. For example, in Figure 5, we have the path-path: $(P4, P2, P1)$. We map each vertex in the path-path to a two-dimensional space as follows. First, all the vertices on the same path have the same path ID, which we define to be vertices' Y labels. For instance, vertices on $P4, P2$ and $P1$ have path ID $1, 2$ and 3, respectively, and their Y values are $1, 2$ and 3, respectively.

Then, we perform a depth-first search (DFS) to create an X label for each vertex (The procedure is sketched in Algorithm 2). In the DFS search, we maintain a counter N, whose initial value equals to the number of all vertices in the graph (In our running example, $N = 13$, see Figure 5). We begin the DFS search with the first vertex v_0 in the first path. In our example, it is the vertex 9 in path $P4$. Starting from this vertex, our DFS search always tries to visit its right neighbor (on the same path) and then tries to visit its upper neighbor (on the path that has the next Y value). For each vertex,

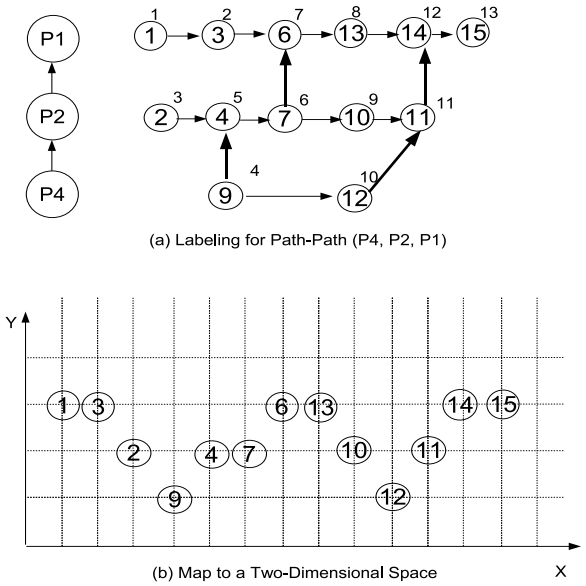


Fig. 5. Labeling for Path-Path (A simple case of Path-Tree)

when we finish visiting all of its neighbors, we set the X label of this vertex to N and reduce N by one. In our example, we start with vertex 9, then visit vertex 12, 11, 14, and 15. Vertex 15 has no right or upper neighbors, so we assign vertex 15 an X label of $N = 13$. Once we visit all the vertices which can be reached from v_0, we start from the first vertex in the second path if it has not been visited. We continue this process until all the vertices have been visited. Note that our labeling procedure bears some similarity to [Kameda 1975]. However, their procedure can handle only a specific type of planar graph, while our labeling procedure handles path-tree graphs which can be non-planar.

Figure 5(a) shows the X label based on the DFS procedure. Figure 5(b) shows the embedding of the path-path in the two dimensional space based on their X and Y labels.

LEMMA 3. *Given two vertices u and v in the path-path, u can reach v if and only if $u.X \leq v.X$ and $u.Y \leq v.Y$ (this is also referred to as u dominates v).*

Proof: First, we prove $u \rightarrow v \Longrightarrow u.X \leq v.X \wedge u.Y \leq v.Y$. Clearly if u can reach v, then $u.Y \leq v.Y$ (path-path property), and DFS traversal will visit u earlier than v, and only after visiting all v's neighbor will it return to u. So, $u.X \leq v.X$ based on DFS. Second, we prove $u.X \leq v.X \wedge u.Y \leq v.Y \Longrightarrow u \rightarrow v$. This can be proved by way of contradiction. Let us assume u cannot reach v. Then, (Case 1:) if u and v are on the same path ($u.Y = v.Y$), then we will visit v before we visit u since u cannot reach v. In other words, we complete u's visit before we complete v's visit. Thus, we get $u.X > v.X$, a contradiction. (Case 2:) if u and v are on different paths ($u.Y < v.Y$), similar to case 1, we will complete the visit of u before we complete the visit of v as u can not reach v. So we have $u.X > v.X$, a contradiction. Combining both cases 1 and 2, we prove our result. □

For the general case, instead of having a single path, the paths form a tree. In this case, each vertex will have an additional interval labeling (see [Agrawal et al. 1989] for details) based on the tree structure. Figure 6(c) shows the interval labeling of the

Algorithm 2 DFSLabel($G[P](V, E), P_1 \cup \cdots \cup P_k$)

Parameter: $P_1 \cup \cdots \cup P_k$ is the path-decomposition of G
Parameter: $G[P]$ is represented as linked lists: $\forall v \in V : linkedlist(v)$ records all the immediate neighbors of v. Let $v \in P_i$. *If v is not the last vertex in path P_i, the first vertex in the linked list is the next vertex of v in the path*
Parameter: $P_i \preceq P_j \Longleftrightarrow i \leq j$

```
1: N ← |V|
2: for i = 1 to k do
3:    v ← P_i[1] {P_i[1] is the first vertex in the path}
4:    if v is not visited then
5:       DFS(v)
6:    end if
7: end for
```

Procedure DFS(v)

```
1: visited(v) ← TRUE
2: for each v' ∈ linkedlist(v) do
3:    if v' is not visited then
4:       DFS(v')
5:    end if
6: end for
7: X(v) ← N {Label vertex v with N}
8: N ← N − 1
```

$\mathcal{SP}$-tree whose corresponding path-tree is shown in Figure 6(a). All the vertices on the path share the same interval label for this path. Besides, the Y label for each vertex is generalized to be the level of its corresponding path in the tree path, i.e., the distance from the path to the root (we assume there is a virtual root connecting all the roots of each tree in the branching/forest). The X label is similar to the simple path-path labeling. The only difference is that each vertex can have more than one upper-neighbor. Besides, we note that we will traverse the first vertex in each path based on the path's level in the $\mathcal{SP}$-tree and any of the traversal orders of the paths in the same level will work for the Y labeling. Figure 6(a) shows the X label of all the vertices in the path-tree and Figure 6(b) shows the two dimensional embedding.

LEMMA 4. *Given two vertices u and v in the path-tree, u can reach v if and only if 1) u dominates v, i.e., $u.X \leq v.X$ and $u.Y \leq v.Y$; and 2) $v.I \subseteq u.I$, where $u.I$ and $v.I$ are the interval labels of u and v's corresponding paths.*

Proof: First, we note that the procedure will maintain this fact that if u can reach v, then $u.X \leq v.X$. This is based on the DFS procedure. Assuming u can reach v, then, there is a path in the tree from u's path to v's path. So we have $v.I \subseteq u.I$ (based on the tree labeling) and $u.Y \leq v.Y$.

On the other hand, if we have $v.I \subseteq u.I$ (which implies $u.Y \leq v.Y$), then there is a path from u's path to v's path. Then, using the similar argument from Lemma 3, we conclude $u.X > v.X$ if u cannot reach v, which implies u can reach v if $u.X \leq v.X$. Proof completes. □

Assuming any interval I has the format $[I.begin, I.end]$, we have the following theorem:

THEOREM 1. *A three dimensional vector labeling $(X, I.begin, I.end)$ is sufficient for answering the reachability query for any path-tree.*

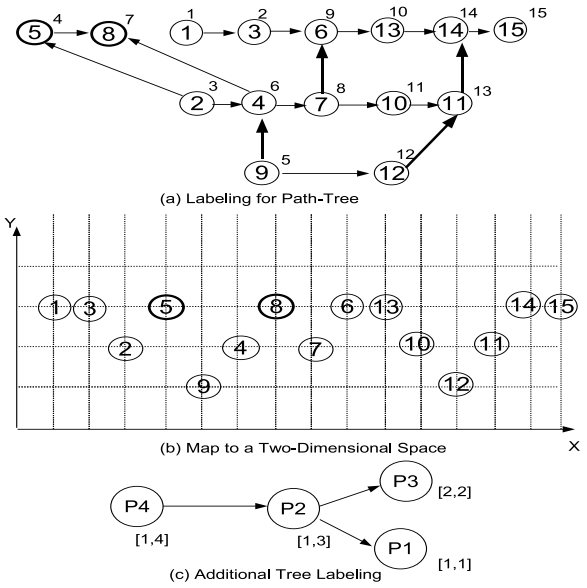


Fig. 6. Complete Labeling for the Path-Tree

Proof: Note that if $v.I \subseteq u.I$, then $v.Y \geq u.Y$. Thus, we can drop Y's label without losing any information. Thus, for any vertex v, we have $v.X$ (the first dimension) and $v.I$ (the interval for the last two dimensions). □

Our labeling algorithm for path-tree is very similar to the labeling algorithm for path-path. It has two steps:

(1) Create tree labeling for the maximum Directed Spanning Tree (maximum $\mathcal{S}P$-tree) obtained from weighed directed path-graph (by Edmonds' algorithm), as shown in Figure 6(c)
(2) Let $P^L = P_1^L \cup \cdots \cup P_{k'}^L$, where P_i^L is the set of vertices (i.e. paths) in level i of the maximum Directed Spanning Tree, which has k' levels. Call Algorithm 2 with $G[P^L](V, E), P_1^L \cup \cdots \cup P_{k'}^L$.

The overall construction time of the path-tree cover is as follows. The first step of path-decomposition is $O(n + m)$, which includes the cost of the topological sort. The second step of building the weighted directed path-graph is $O(m)$. The third step of extracting the maximum spanning tree is $O(m' + k \log k)$, where $m' \leq m$ and $k \leq n$. The fourth step of labeling basically utilizes a DFS proceduce which costs $O(m'' + n)$, where m'' is the number of edges in the path-tree and $m'' \leq m$. Thus, the total construction time of path-tree cover is $O(m + n \log n)$.

2.5. Step 5: Transitive Closure Compression and Reachability Query Answering

Edges not included in the path-tree cover can result in extra reachability which will not be covered by the path-tree structure. Similar problem appears in the tree cover related approaches.

For example, Dual Labeling and GRIPP utilize a tree as their first steps; they then try to find novel ways to handle non-tree edges. Their ideas are in general applicable to dealing with non-path-tree edges as well. From this perspective, our path-tree cover approach can be looked as being orthogonal to these approaches.

To answer a reachability query for the entire DAG, a simple strategy is to actually construct the transitive closure for non-path-tree edges in the DAG. The construction time is $O(n+m+t'^3)$ and the index size is $O(n+t'^2)$ according to Dual Labeling [Wang et al. 2006], where t' is the number of non-path-tree edges. However, as we will see later in theorem 5, if the path-tree cover approach utilizes the same tree cover as Dual Labeling for a graph, t' is guaranteed to be smaller than t (non-tree edges).

Moreover, if a maximally compressed transitive closure is desired, the path-tree structure can help us significantly reduce the transitive size (index size) and its construction time as well. Let $R^c(u)$ be the compressed set of vertices we record for u's transitive closure utilizing the path-tree. Assume u is a vertex in P_i. To answer a reachability query for u and v (i.e. if v is reachable from u), we need to test 1) if v is reachable from u based on the path-tree labeling and if not 2) for each x in $R^c(u)$, whether v is reachable from x based on the path-tree labeling. We note that $R^c(u)$ includes at most one vertex from any path and in the worst case, $|R^c(u)| = k$, where k is number of paths in the path-tree, i.e. number of vertices in $\mathcal{SP}$-tree. Thus, a query would cost $O(k)$. In Subsection 3.4, we will introduce a procedure which costs $O(\log^2 k)$.

Algorithm 3 is an easily understandable algorithm for transitive closure construction. To construct transitive closure for each vertex, we call Algorithm 3 with $j = 1$. The step 9 of algorithm 3 will be called $O(mk)$ times because for any vertex u, $|R^c(u)| \leq k$. Later in Section D, we will call Algorithm 3 again with j as a starting number for partially reconstructing transitive closure.

To obtain the minimum transitive closure, in step 9 of Algorithm 3, we add v' into $R^c(V_R[i])$ using the following rule, so that there will not exist two vertices in $R^c(V_R[i])$ such that one can reach another through $G[T]$. Thus, the time complexity of algorithm 3 is $O(mk^2)$.

ADDING RULE
IF $\exists u \in R^c(V_R[i])$ such that $u \rightarrow v'$ in $G[T]$
Discard v' and Return;
ENDIF
IF $\exists u \in R^c(V_R[i])$ such that $v' \rightarrow u$ in $G[T]$
Delete u from $R^c(V_R[i])$;
ENDIF
Insert v' into $R^c(V_R[i])$.

Algorithm 3 CompressTransitiveClosure (G,$G[T]$,j)

1: $V_R \leftarrow Reversed\ Topological\ Order\ of\ G$ {Perform topological sort of G}
2: $N \leftarrow |V|$
3: **for** $i = j$ to N **do**
4: $R^c(V_R[i]) \leftarrow \emptyset$;
5: Let S be the set of immediate successors of $V_R[i]$ in G;
6: **for each** $v \in S$ **do**
7: **for each** $v' \in R^c(v) \cup \{v\}$ **do**
8: **if** $V_R[i]$ cannot reach v' in $G[T]$ **then**
9: Add v' into $R^c(V_R[i])$;
10: **end if**
11: **end for**
12: **end for**
13: **end for**

However, a careful redesign of algorithm 3 can achieve $O(mk)$ time complexity. In algorithm 3, we merge the transitive closure (including the vertex itself) of each immediate successor v of $V_R[i]$, i.e. $R^c(v) \cup \{v\}$, into the transitive closure of $V_R[i]$, i.e. $R^c(V_R[i])$. The merge takes $O(k^2)$ because we compare each vertex in $R^c(v) \cup \{v\}$ with each vertex in $R^c(V_R[i])$.

Such merge can be done in $O(k)$ time if we organize the transitive closure $R^c(u)$ of any vertex u such that vertices in $R^c(u)$ are ordered according to their paths. Merging two transitive closures involves only comparisons between vertices of the same path. If there exists two vertices v and u belonging to the same path, i.e., $v.I == u.I$, we only keep the vertex with the smaller X label. For example, if $v.X < u.X$, then we only keep v because v can reach u on the path-tree.

After merge the transitive closure of a $V_R[i]$'s immediate successor by the above method, the transitive closure $R^c(V_R[i])$ may not be optimal, i.e. there may exists two vertex x and y in $R^c(V_R[i])$ such that x can reach y through $G[T]$ and thus y is redundant. To optimize the transitive closure, we can depth-first traverse $\mathcal{SP}$-tree, which is a spanning tree (with k vertices) of the path-graph. When we visit the first path P_x which contains a vertex x in $R^c(V_R[i])$, we put x in a stack s. Later when we visit another path P_y which is a descendent of P_x in $\mathcal{SP}$-tree, and contains a vertex y in $R^c(V_R[i])$, we remove y from $R^c(V_R[i])$ if (1) x (i.e. the first vertex in the stack s) can reach y through $G[T]$, or we push y into the stack s if (2) x (i.e. the first vertex in the stack s) cannot reach y through $G[T]$.

In case (1), if y can reach another vertex z through $G[T]$ then x can also reach z through $G[T]$ and thus y is redundant.

In case (2), it is easy to see the X label of vertex y is smaller than the X label of vertex x. If through $G[T]$, x can reach another vertex z in path P_z, which is a descendant of P_y in $\mathcal{SP}$-tree, y can also reach z through $G[T]$. This implies that by comparing only with the top element on the stack s, we will not fail to identify a redundant vertex.

We depth-first traverse the tree according to above rule and remove the top vertex, i.e., the vertex w associated with P_w, from the stack s when we finish visiting path P_w and its descendant in $\mathcal{SP}$-tree.

Since depth-first traverse takes $O(k)$ time, we conclude the above procedure of optimizing transitive closure takes $O(k)$ time for each vertex, and the total construction time of path-tree takes $O(mk + nk) = O(mk)$ time in worst case.

Algorithm 4 gives the complete pseudocode of the improved transitive closure construction which takes $O(mk)$ time in worst case.

3. THEORETICAL ANALYSIS OF OPTIMAL PATH-TREE COVER CONSTRUCTION

In this section, we investigate several theoretical issues related to path-tree cover construction. We show that given the path-decomposition of DAG G, the problem of finding the optimal path-tree cover of G is equivalent to that of finding the *maximum spanning tree* of a directed graph. We demonstrate that the optimal tree cover by Agrawal *et al.* [Agrawal et al. 1989] is a special case of our problem. In addition, we show that our path-tree cover can always achieve better compression than any chain cover or tree cover.

To achieve this, we utilize the predecessor set of each vertex. But first we note that the computational cost for computing all of the predecessor sets of a given DAG G is equivalent to the cost of the transitive closure of G, with $O(nm)$ time complexity. Thus it may not be applicable to very large graphs as its computational cost would be prohibitive. It can, however, still be utilized as a starting point for understanding the potential of path-tree cover, and its study may help to develop better heuristics to efficiently extract a good path-tree cover. In addition, these algorithms might be better

Algorithm 4 FastCompressTransitiveClosure(G,$G[T]$,j)

1: $V_R \leftarrow Reversed\ Topological\ Order\ of\ G$ {Perform topological sort of G}
2: $N \leftarrow |V|$
3: **for** $i = j$ to N **do**
4: $R^c(V_R[i]) \leftarrow \emptyset$;
5: Let S be the set of immediate successors of $V_R[i]$ in G;
6: **for each** $v \in S$ **do**
7: Merge $(\mathcal{R}^c(v) \cup \{v\})$ into $R^c(V_R[i])$; {Merging takes $O(k)$ time. After merging, no two vertices in $R^c(V_R[i])$ belong to the same path}
8: **end for**
9: Mapping all vertices in $R^c(V_R[i])$ to their corresponding vertex in $\mathcal{SP}$-tree;
10: DFSCompress($\mathcal{SP}$-tree, root-of-$\mathcal{SP}$-tree, $\emptyset$, $R^c(V_R[i])$); {Recursively remove redundant vertices in $R^c(V_R[i])$ in $O(k)$ time}
11: **end for**

Procedure DFSCopmress($\mathcal{SP}$, r, s, R)
Parameter: Maximum Spanning Tree $\mathcal{SP}$, Vertex r (in $\mathcal{SP}$), Stack s, Closure R

1: **if** r associates with a vertex u in R **then**
2: **if** $s.top() \rightarrow u$ {$s.top()$ reach u in Path-Tree Cover} **then**
3: $R \leftarrow R \backslash \{u\}$ {Drop vertex u in closure R}
4: **else**
5: $s.push(u)$;
6: **for each** $r' \in children(r)$ **do**
7: DFSCompress($\mathcal{SP}, r', s, R$);
8: **end for**
9: $s.pop()$;
10: **end if**
11: **end if**

suited for other applications if the knowledge of predecessor sets is readily available. Thus, they can be applied to compress the transitive closure.

We will introduce an optimal query procedure for reachability queries which can achieve $O(\log^2 k)$ time complexity in the worst case, where k is the number of paths in the path-decomposition.

Due to the space limitation, some proofs in this section are included in the Appendix.

3.1. Optimal Path-Tree Cover with Path-Decomposition

We first consider the following restricted version of the optimal path-tree cover problem.

Optimal Path-Tree Cover (OPTC) Problem: *Let $P = (P_1, \cdots, P_k)$ be a path-decomposition of DAG G, and let $\mathcal{G}_s(P)$ be the family including all the path tree covers of G which are based on P. The OPTC problem tries to find the optimal tree cover $G[T] \in \mathcal{G}_s(P)$, such that it requires minimum index size to compress the transitive closure of G.*

To solve this problem, let us first analyze the index size which will be needed to compress the transitive closure utilizing a path-tree $G[T]$. Note that $R^c(u)$ is the compressed set of vertices which vertex u can reach and for compression purposes, $R^c(u)$ does not include v if 1) u can reach v through the path-tree $G[T]$ and 2) there is an vertex $x \in R^c(u)$, such that x can reach v through the path-tree $G[T]$. Given this, we

can define the compressed index size as

$$Index\_cost = \sum_{u \in V(G)} |R^c(u)|$$

(We omit the labeling cost for each vertex as it is the same for any path-tree.) To optimize the index size, we utilize the following equivalence.

LEMMA 5. *For any vertex v, let $T_{pre}(v)$ be the **immediate** predecessor of v on the path-tree $G[T]$. Then, we have*

$$Index\_cost = \sum_{v \in V(G)} |S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))|$$

where $S(v)$ includes all the vertices which can reach vertex v in DAG G.

Given this, we can solve the OPTC problem by utilizing the predecessor sets to assign weights to the edges of the weighted directed path-graph in Subsection 2.3. Thus, the path-tree which corresponds to the maximum spanning tree of the weighted directed path-graph optimizes the index size for the transitive closure. Consider two paths P_j, P_i and the minimal equivalent edge set $E^R_{P_j \to P_i}$. For each edge $(u, v) \in E^R_{P_j \to P_i}$, let v' be the vertex which is the immediate predecessor of v in path P_i. Then, we define the predecessor set of v with respect to u as

$$S_u(v) = (S(u) \cup \{u\}) \backslash (S(v') \cup \{v'\})$$

If v is the first vertex in the path P_j, then we define $S(v') = \emptyset$. Given this, we define the weight from path P_j to path P_i as

$$w_{P_j \to P_i} = \sum_{(u,v) \in E^R_{P_j \to P_i}} |S_u(v)|$$

We refer to such criteria as *OptIndex*.

THEOREM 2. *The path-tree cover corresponding to the maximum spanning tree from the weighted directed path-graph defined by OptIndex achieves the minimum index size for the compressed transitive closure among all the path-trees in $\mathcal{G}_s(P)$.*

Proof: We decompose $Index\_cost$ utilizing the path-decomposition $P = P_1 \cup \cdots \cup \cdots P_k$ as follows:

$$Index\_cost = \sum_{1 \le i \le k} \sum_{v \in P_i} |S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))|$$

Note that $S(v) \supseteq (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$. Then, minimizing the $Index\_cost$ is equivalent to maximizing

$$\sum_{1 \le i \le k} \sum_{v \in P_i} | \bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\})|$$

Recalling T (let its directed edge set be $E(T)$) is the $\mathcal{SP}$-tree (i.e. the spanning tree for the weighted directed path-graph), we can further rewrite the above expression as (v_l being the vertex with largest sid in the path P_i)

$$\sum_{1 \le i \le k} (\sum_{v \in P_i \backslash \{v_l\}} |S(v) \cup \{v\}| + \sum_{(u,v) \in E^R_{P_j \to P_i}, 1 \le j \le k, (j,i) \in E(T)} |S_u(v)|)$$

where P_j is the parent path in the path-tree of path P_i. Since the first half of the sum is the same for the given path decomposition, we essentially need to maximize

$$\sum_{1\le i\le k}\ \sum_{(u,v)\in E^R_{P_j\to P_i},1\le j\le k,(j,i)\in E(T)} |S_u(v)| = \sum_{1\le i\le k,1\le j\le k,(i,j)\in E(T)} w_{P_j\to P_i}$$

Maximum spanning tree T of the weighted directed path-graph defined by OptIndex will maximize $\sum_{1\le i\le k,1\le j\le k,(i,j)\in E(T)} w_{P_j\to P_i}$ and thus the theorem holds. □

Recall that in Agrawal's optimal tree cover algorithm [Agrawal et al. 1989], to build the tree, for each vertex v in DAG G, essentially they choose its immediate predecessor u with the maximal number of predecessors as its parent vertex, i.e.,

$$|S(u)| \ge |S(x)|, \forall x \in in(v), u \in in(v)$$

Given this, we can easily see that if the path decomposition treats each vertex in G as an individual path, then we have the optimal tree cover algorithm from Agrawal *et al.* [Agrawal et al. 1989].

THEOREM 3. *The optimal tree cover algorithm [Agrawal et al. 1989] is a special case of path-tree construction when each vertex corresponds to an individual path and the weighted directed path-graph utilizes the* OptIndex *criteria.*

Proof: By definition of OptIndex,

$$w_{P_j\to P_i} = \sum_{(u,v)\in E^R_{P_j\to P_i}} |S_u(v)|$$

When each vertex corresponds to an individual path, it's easy to see the weight on the edge $(u, v) \in E(G)$ in the weighted directed path-graph (each path is a single vertex) is

$$w_{u,v} = \sum |S(u) \cup \{u\}|$$

, which is exactly the size of the predecessor set of u plus u itself.

In this case, optimal tree cover algorithm, in which each vertex selects the vertex (in its immediate predecessor set) with maximum predecessor set as its father, works equally as path-tree construction, which maximizes the weighted directed path-graph. Proof completes. □

3.2. Optimal Path-Decomposition

Theorem 2 shows the optimal path-tree cover for the given path-decomposition. A follow-up question is then how to choose the path-decomposition which can achieve overall optimality. This problem, however, remains open at this point (undecided between P and NP). Instead, we ask the following question.

Optimal Path-Decomposition (OPD) Problem: *Assuming we utilize only the path-decomposition to compress the transitive closure (in other words, no cross-path edges), the OPD problem is to find the optimal path-decomposition which can maximally compress the transitive closure.*

There are clearly cases where the optimal path-decomposition does not lead to the perfect path-tree that *alone* can answer all the reachability queries. This nevertheless provides a good heuristic to choose a good path-decomposition in the case where the predecessor sets are available. Note that the OPD problem is different from the chain decomposition problem in [Jagadish 1990], where the goal is to find the minimum width of the chain decomposition.

We map this problem to the *minimum-cost flow* problem [Goldberg et al. 1990]. We transform the given DAG G into a network G_N (referred to as the flow-network for G)

as follows. First, each vertex v in G is split into two vertices s_v and e_v, and we insert a single edge connecting s_v to e_v. We assign the cost of such an edge $F(s_v, e_v)$ to be 0. Then, for an edge (u, v) in G, we map it to (e_u, s_v) in G_N. The cost of such an edge is $F(e_u, s_v) = -|S(u) \cup \{u\}|$, where $S(u)$ is the predecessor set of u. Finally, we add a virtual source vertex and a virtual sink vertex. The virtual source vertex is connected to any vertex s_v in G_N with cost 0. Similarly, each vertex e_v is connected to the sink vertex with cost being zero. The capacity of each edge in G_N is one ($C(x, y) = 1$). Thus, each edge can take maximally one unit of flow, and correspondingly each vertex can belong to one and only one path.

Let $c(x, y)$ be the amount (0 or 1) of flow over edge (x, y) in G_N. The cost of the flow over the edge is $c(x, y) \cdot F(x, y)$, where $c(x, y) \leq C(x, y)$. We would like to find a set of flows which go through all the vertex-edges (s_v, e_v) and whose overall cost is minimum. We can solve it using an algorithm for the *minimum-cost flow* problem for the case where the amount of flow being sent from the source to the sink is given. Let i-flow be the solution for the minimum-cost flow problem when the total amount of flow from the source to the sink is fixed at i units. We can then vary the amount of flow from 1 to n units and choose the largest one i-flow which achieves the minimum cost. It is apparent that i-flow goes through all the vertex-edges (s_v, e_v).

THEOREM 4. *Let G_N be the flow-network for DAG G. Let F_k be the minimum cost of the amount of k-flow from the source to the sink, $1 \leq k \leq n$. Let i-flow from the source to the sink minimize all the n-flow, $F_i \leq F_k, 1 \leq k \leq n$. The i-flow corresponds to the best index size if we utilize only the path-decomposition to compress the transitive closure.*

Proof: First, we can prove that for any given i-flow, where i is an integer, the flow with minimum-cost will pass each edge either with 1 or 0 (similar to the Integer Flow property [Cormen et al. 2001]). Basically, the flow can be treated as a binary flow. In other words, any flow going from the source to the sink will not split into branches (note that each edge has only capacity one). Thus, applying Theorem 2, we can see that the total cost of the flow (multiplied by negative one) corresponds to the savings for the *Index_cost*

$$\sum_{1 \leq i \leq k} \Big(\sum_{v \in P_i \setminus \{v_l\}} |S(v) \cup \{v\}| \Big) = \sum_{(u,v) \in G_N} c(u, v) \times F(u, v)$$

where v_l is the vertex with largest sid in path P_i. Then, let i-flow be the one which achieves minimum cost (the most negative cost) from the source to the sink. Thus, when we invoke the algorithm which solves the *minimum-cost maximum flow*, we will achieve the minimum cost with i-flow. It is apparent that the i-flow goes through all the vertex-edges (s_v, e_v) because i is largest. Thus, we identify the flow and find our path-decomposition. □

Note that there are several algorithms which can solve the minimum-cost maximum-flow problem with different time complexities. Interested readers can refer to [Goldberg et al. 1990] for more details. Our methods can utilize any of these algorithms.

3.3. Superiority of Path-Tree Cover Approach

For any tree cover, we can also transform it into a path-decomposition. We extract the first path by taking the shortest path from the tree cover root to any of its leaves. After we remove this path, the tree will then break into several subtrees. We perform the same extraction for each subtree until each subtree is empty. Thus, we can have the path-decomposition based on the tree cover. In addition, we note that there is at most one edge linking two paths.

Given this, we can prove the following theorem.

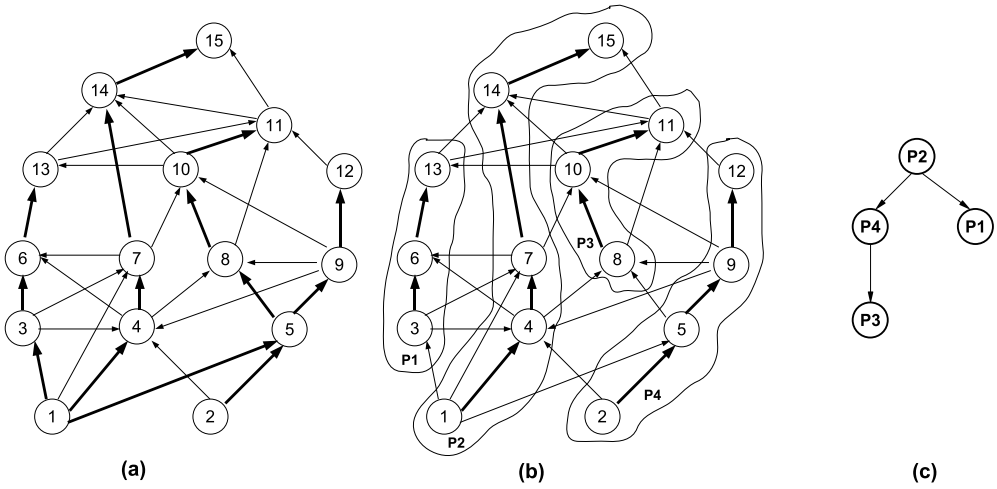


Fig. 7. (a) a spanning tree for a DAG. There is a virtual root connecting vertex 1 and 2. (b) A path decomposition from the spanning tree. (c) A $\mathcal{SP}$-tree as a result of (a) and (b).

THEOREM 5. *For any tree cover, the path-tree cover which uses the path-decomposition from the tree cover and is built by OptIndex has a transitive closure size lower than or equal to the transitive closure size of the corresponding tree cover.*

Proof: This follows Theorem 2. By decomposing the spanning tree into a set of paths, we can convert a tree cover into a path-tree cover which can represent the same shape as the original tree cover, i.e., if there is an edge in the original tree cover linking path P_i to P_j, there is a path-tree that can preserve its path-path relationship by adding this edge and possibly more edges from path P_i to P_j in DAG G to the path-tree.

By theorem 2, we can get a path-tree cover of minimum index size (i.e. index size no larger than the path-tree cover converted directly from tree cover) given the set of paths as a decomposition from the spanning tree. □

Figure 7 shows an example of converting a tree cover into a path-tree cover. Figure 7(a) is a spanning tree for a DAG, assuming there is a virtual root connecting vertex 1 and 2. By decomposing the spanning tree of (a) into paths, we get a path decomposition of the DAG in Figure 7(b).

Finally, we can construct a $\mathcal{SP}$-tree as shown in Figure 7(c), by taking into consideration of the spanning tree in (a), and the path decomposition the (b). Consider an edge $(1, 3)$ in the original spanning tree of (a), which is the edge from $P2$ to $P1$ in (b), its corresponding edge in the $\mathcal{SP}$-tree of (c) is $(P2, P1)$. It's easy to see that the $\mathcal{SP}$-tree edge $(P2, P1)$ contains more information than the edge $(1, 3)$ in the original spanning tree.

In addition, one can also conclude that the conversion of a tree cover into a path-tree cover is not unique.

3.4. Very Fast Query Processing

Given source vertex u and destination vertex v, query processing will answer whether u can reach v by checking

— (1) whether u can reach v (or $u = v$) by path-tree cover.

— (2) whether any vertex w in $R^c(u)$ can reach v (or $w = v$) by path-tree cover.

u can reach v if and only if (1) or (2) holds.

Here we describe an efficient query processing procedure algorithm with $O(\log^2 k)$ query time, where k is the number of paths in the path-decomposition. We first build an interval tree based on the intervals of each vertex in $R(u)$ as follows (this is slightly different from "Section 10.1 Interval Trees" of [de Berg et al. 2008]): Let x_{mid} be the median of the end points of the intervals. Let $w.I.begin$ and $w.I.end$ be the starting point and ending point of the interval $w.I$, respectively. We define the interval tree recursively. The root node r of the tree stores x_{mid}, such that

$$\begin{aligned} \mathcal{I}_{left} &= \{w | w.I.end < x_{mid}\} \\ \mathcal{I}_{mid} &= \{w | x_{mid} \in w.I\} \\ \mathcal{I}_{right} &= \{w | w.I.begin > x_{mid}\} \end{aligned}$$

Then, the left subtree of r is an interval tree for the set of $\mathcal{I}_{left}$ and the right subtree of r is an interval tree for the set of $\mathcal{I}_{right}$. We store $\mathcal{I}_{mid}$ only once and order it by $w.X$ (the X label of the vertices in $\mathcal{I}_{mid}$). This is the key difference between our interval tree and the standard one from [de Berg et al. 2008]. In addition, when $\mathcal{I}_{mid} = \emptyset$, the interval tree is a leaf. Following [de Berg et al. 2008], we can easily construct in $O(k \log k)$ time an interval tree using $O(k)$ storage and depth $O(\log k)$.

Algorithm 5 Query(r,v)

```
1: if r is not a leaf then
2:    Perform BinarySearch to find a vertex w on r.I_mid whose w.X is the closest one
      such that w.X ≤ v.X
3: else
4:    return FALSE;
5: end if
6: if v.I ⊆ w.I then
7:    return TRUE;
8: end if
9: if v.I.end < r.x_mid then
10:   Query(r.left,v);
11: else
12:   if v.I.begin > r.x_mid then
13:      Query(r.right,v);
14:   end if
15: end if
```

The query procedure using this interval tree is shown in Algorithm 5 *when u does not reach v through the path-tree*. Line 9 to 15 functions as a binary search and Query(r,v) will be called $O(\log k)$ times because the interval tree is a balanced tree with at most k nodes. Line 2 is a binary search which takes $O(\log k)$ time because the number of vertices in $r.\mathcal{I}_{mid}$ is no more than k. Hence, we conclude the time complexity of the algorithm is $O(\log^2 k)$. The correctness of the Algorithm 5 can be proved as follows.

LEMMA 6. *For the case where u cannot reach v using only the path-tree, then Algorithm 5 correctly answer the reachability query.*

4. GENERALIZING PATH-TREE TO CHAIN-TREE

4.1. Chain-Tree Definition and its Construction

A *chain* of a DAG is a sequence of vertices $(v_0, v_1, \ldots, v_p)$ in which v_{i+1} is reachable from v_i $(0 \leq i \leq p-1)$. It is a generalization of path. A major difference between chain decomposition and path-decomposition is that each path P_i in the path-decomposition is a subgraph of G. However, a chain may not be a subgraph of G. It is a subgraph of the transitive closure of G. Similarly, *a* **chain-tree** *on G can be defined as a path-tree on $G' = (V, E')$ which is the transitive closure graph of $G = (V, E)$, i.e., for any two vertices x and y in V, there exists a directed edge (x, y) in E' if and only if x can reach y in G.*

Recall that there are five steps (detailed in Section 2) to construct a path-tree index for G, 1) path decomposition, 2) pair-path subgraph construction, 3) path-tree cover extraction from path-graph, 4) reachability labeling for path-tree cover, and 5) transitive closure compression using path-tree cover. To construct the chain-tree cover and use it to compress transitive closure, the straightforward way is simply based on the chain-tree definition, i.e., we can simply compute the full transitive closure $G' = (V, E')$, where E' include all reachable vertex pairs (x, y), $x \rightarrow y$, and then perform the above five steps in building the chain-tree indexing. Clearly, this can be very expensive due to the explicit construction and processing of the full transitive closure G'. Given this, the key questions are how to construct chain-tree cover without G'. Specifically, we need to reconsider the first two steps: 1) chain decomposition, and 2) pair-chain subgraph construction. Once we complete these two steps, the other three steps will remain the same as the path-tree indexing.

(Step 1) Chain Decomposition: In this step, we consider to directly perform chain decomposition of G.

DEFINITION 4. *Let $G = (V, E)$ be a DAG. We say a partition $C_1, \cdots, C_k$ of V is a chain-decomposition of G if and only if $C_1, \cdots, C_k$ are chains of G, and $C_1 \cup \cdots \cup C_k = V$, and $C_i \cap C_j = \emptyset$ for any $i \neq j$. We also refer to k as the width of the decomposition.*

Similar to path tree construction, it is still an open problem on how to choose the best chain decomposition to construct the optimal chain-tree which can maximally compress the transitive closure. A well studied problem is to compute the minimum chain decomposition, i.e. a chain decomposition with minimum number of chains. Although minimum number of chains by no means implies minimum size of transitive closure, it nevertheless provides a good heuristic for construct a chain-tree.

In the area of order theory, the partial order width of any partially ordered set (i.e. DAG) is defined as the size of the longest antichain in the partially ordered set. Dilworth's Lemma [Dilworth 1950] shows that the partial order width of a DAG is equal to the minimum number of chains needed to cover the DAG. König's theorem [König 1884] shows that in bipartite graphs maximum matching problem and the minimum vertex cover problem are the same. The connection between minimum chain cover (i.e. minimum chain decomposition) and bipartite matching allows the minimum chain decomposition to be computed in polynomial time by maximum flow algorithms [Cormen et al. 2001]. But worst case time complexity of any available maximum flow algorithms, to the best of our knowledge, is no smaller than that of transitive closure computation.

(Step 2) Pair-Chain Subgraph: Once we have the chain decomposition, we need consider *pair-chain* subgraphs, $C_i \rightarrow C_j$ consisting of i) chain C_i, ii) chain C_j, and iii)$E'_{C_i \rightarrow C_j}$, which contains all the vertex pairs (x, y), such that $x \in C_i$, $y \in C_j$, and $x \rightarrow y$. In other words, $C_i \rightarrow C_j \subseteq G'$. Importantly, we can simplify $E'_{C_i \rightarrow C_j}$ by removing all the vertex pairs whose removal does not change the reachability information from

C_i to C_j. The simplified edge set, denoted as $E'^R_{C_i \to C_j}$, is the counterpart of *minimal equivalent edge set* (Definition 2) for the pair-path subgraph.

Clearly, to compute $E'^R_{C_i \to C_j}$, the straightforward method is simply using Algorithm 1 on the full transitive closure G'. Interestingly, we note that the minimal equivalent edge set $\cup E'^R_{C_i \to C_j}$ on G' actually corresponds to the set of *contour points* in [Jin et al. 2009]. Basically, an edge $(x, y) \in \cup E'^R_{C_i \to C_j}$ iff (x, y) is a contour point of G. Specifically, in [Jin et al. 2009], authors provide a $O(mk \log n)$ algorithm to discover all the contour pairs given a chain decomposition of k chains without materializing the full transitive closure. Thus, we can employ their method here to construct the minimal equivalent edge set for each pair-chain graph.

4.2. Optimality of Chain-Tree

It is easy to see that in general an optimal chain-tree is always not worse than an optimal path-tree on a DAG G because any path-tree is a special case of chain-tree.

In the following, we will prove an even stronger result by considering building the chain-tree based on an existing path decomposition. Note that different from the path-tree, the chain-tree considers all the reachability vertex pairs between two paths (pair-chain subgraph), whereas the path-tree only considers the edges between two paths (pair-path subgraph).

LEMMA 7. *Given two paths P_i and P_j, the weight from path P_j to P_i on G, i.e., $w_{P_j \to P_i} = \sum_{(u,v) \in E^R_{P_j \to P_i}} |S_u(v)|$ (See $S_u(v)$ definition in Section 3.1), is no larger than the weight from path P_j to P_i on G', $w'_{P_j \to P_i} = \sum_{(u,v) \in E'^R_{P_j \to P_i}} |S_u(v)|$, i.e. $w'_{P_j \to P_i} \geq w_{P_j \to P_i}$.*

The proof of this lemma can be found in the Appendix. Given Lemma 7 and Theorem 2, we can easily see the following results.

THEOREM 6. *Given a path decomposition, an optimal chain-tree is always not worse than an optimal path-tree in compressing the transitive closure of G.*

Note that Theorem 6 provides the following important implication for chain-tree construction: instead of using the expensive chain-decomposition, we can simply apply the existing path-decomposition method to build the chain-tree indexing. Since in Step 2, we consider all the reachable vertex pairs in the pair-chain subgraph construction, Theorem 6 theoretically guarantees that the chain-tree indexing is no worse than the path-tree index with the same path-decomposition.

In fact, given a path-decomposition of G, an optimal chain-tree could be any times better than an optimal path-tree in compressing the transitive closure of G. Figure 8 shows such an example. Edge weights of path-graph are marked in Figure 8(b) and (c). Please recall Section 3.1, particularly the $OptIndex$ criteria, for edge weight calculation. It it easy to figure out, according to Algorithm 3, that the size of transitive closure after the optimal path-tree compression is $y * (k-2)$, while it is $(k-1)(k-2)$ after the optimal chain-tree compression. Thus, if we increase y (and x correspondingly), we can make the optimal chain-tree any times better than the optimal path-tree.

5. POST PROCESSING BY POST-LABELING COMPRESSION

In this section, we present a post-processing framework to further compress the transitive closure and still enable fast reachability query answering. Our idea is to explore the "reachability-similarity" among vertices in graph G. Recall that in the path-tree (chain-tree) indices, each vertex u in graph G records the compressed transitive closure $R^c(u)$, which includes all those vertices u can reach except those reachable from

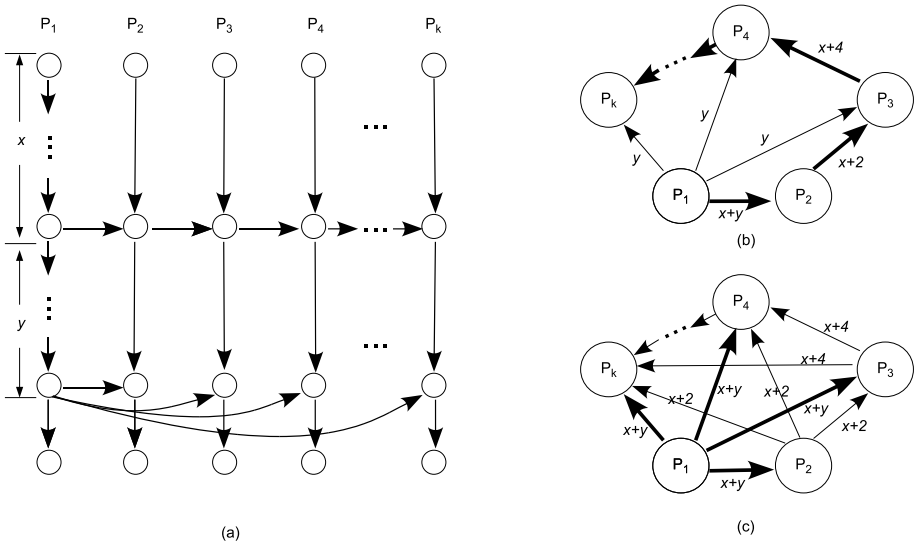


Fig. 8. An example showing an optimal chain-tree can be any times better than an optimal path-tree in compressing the transitive closure of G, given the path-decomposition. (a) The original graph G with path-decomposition ($x \geq y > 2(k-2)$). (b) path-graph and $\mathcal{SP}$-tree for the optimal path-tree. Edge weights are marked on each edge. (c) path-graph and $\mathcal{SP}$-tree for the optimal chain-tree. Edge weights are marked on each edge.

other vertex in $R^c(u)$ in path-tree cover(Subsection 2.5). Given two vertices u and v, if their compressed transitive closures are similar to each other, i.e., $R^c(u) \approx R^c(v)$, can we utilize such property to further reduce the index size without sacrificing much query processing time? Note that similar ideas have been widely studied in graph compression research [Adler and Mitzenmacher 2002; Raghavan and Garcia-Molina 2003; Navlakha et al. 2008] but to the best of our knowledge, are not studied in the reachability indexing.

Indeed, if we consider $R^c(u)$ as the adjacent list of vertex u, then, the compressed transitive closure naturally introduce a new graph G^c. We can equivalently represent the compressed transitive closure using a binary matrix $M = \{s_i, 1 \leq i \leq n\}$ with size $n \times n$, such that for each row s_i (corresponding to vertex u_i in G) encodes $R^c(u)$. Given this, it may seem to be straightforward to apply graph compression techniques to compress G^c. However, those methods typically focus on minimizing the graph representation cost, and do not consider how to efficiently answer query in the compressed graph. Especially, in our reachability indexing framework, the compression has to allow a fast recovery of entire adjacent list $R^c(u)$. For instance, reference encoding [Adler and Mitzenmacher 2002] is one of the well-known methods in graph compression which utilize such adjacency list similarity and most of the available methods stem from it: if vertices x and y have similar adjacency list, then it tries to compress the adjacency list of y by representing it in terms of the adjacency list of x (x is the reference vertex of y). In [Adler and Mitzenmacher 2002], Adler and Mitzenmacher propose to utilize a directed maximal spanning tree to represent such references. Basically, each vertex in the tree will refer to its parent vertex for adjacency list references. The issue is that in order to recover the adjacency list of a vertex u, we have to go through all its ancestors

to the root vertex in the tree. Clearly, this type of reference encoding is prohibitive for reachability query answering.

In this paper, we propose a two-level encoding framework to further condense the compressed transitive closure $R^c(u)$, and we utilize kmeans clustering and a sliding-window algorithm to automatically determine the initial grouping and the number of clusters for kmeans algorithm.

5.1. Two-Level Encoding and its Computation

Two-Level Encoding Framework. The two-level encoding framework is designed to compactly store the compressed transitive closure $R^c(u)$ which can also be unfolded efficiently. In the first level, we partition all the vertices in graph G into groups ($\cup g_i = V$ and $g_i \cap g_j = \emptyset$) and each group g_i have a representative transitive closure, denoted as $R(g_i)$. In the second level, the compressed transitive closure of each vertex u, $R^c(u)$ is encoded in a triple $< g_i, R_+(u), R_-(u) >$, where g_i is the group ID, pointing to the group representative transitive closure $R(g_i)$, $R_+(u)$ records the vertices in $R^c(u)$ but not in $R(g_i)$, and $R_-(u)$ records the vertices in $R(g_i)$ but not in $R^c(u)$. In other words, we have $R(g_i) \cup R_+(u) \backslash R_-(u) = R^c(u)$. Using this two level encoding and this relationship, we can easily recover $R^c(u)$ in $O(k)$ time complexity, where k is the number of paths or chains. In addition, we note this encoding will not help compress those vertices if their compressed transitive closure is already small. Thus, we only apply this additional encoding for any vertex u with $|R^c(u)| \geq 3$.

Kmeans Clustering. Computing the optimal encoding can be transformed to a clustering problem which can be solved efficiently by kmeans. First, we formally define the encoding cost as follows:

$$cost = \sum_{k=1}^{K} \sum_{u \in g_k} H(R^c(u), R(g_i)) + \sum_{k=1}^{K} |R(g_i)|$$

where K is the number of groups and $H(R^c(u), R(g_i))$ denotes the Hamming distance between compressed transitive closure $R^c(u)$ and the group representative $R(g_i)$. The second term corresponds to the cost of representatives in the first level. Since the second term is generally rather small compared with the first term, we can focus on the first term. Furthermore, if we consider the binary vector representation of $R^c(u)$ and $R(g_i)$, it is easy to see optimizing the cost can be directly solved by a binary kmeans procedure which iteratively performs two steps: cluster assignment and centroids update. Based on the cost function, cluster assignment is easily done (simply based on the shortest Hamming distance), thus we focus on the step of updating centroids. Since distance measure is defined on the top of Hamming distance, the majority rule can be applied such that one element exists on the centroid if and only the number of vertices in the group containing this element is no less than half.

The issue of kmeans is that it is rather sensitive to the initial grouping and the number of clusters. In the following, we introduce a new method based on the direct graph structure to handle both issues efficiently and effectively.

A Sliding-Window Initialization Approach. Recall that we generate the (compressed) transitive closure in the reversed topological order. Thus, if we organize the vertices in the same order, it can be expected that some consecutive or "close" transitive closures are very similar to each other. Intuitively, the larger and denser the graph is, more significant locality phenomena would be, because much reachability information should be preserved during propagation in the reversed topological sort order.

Our initialization algorithm is based on such observation. It tries to greedily group several consecutive vertices (closures) together using a sliding-window. Specially, given

the current sliding-window $W = \{u_i, a \leq i \leq (a+x-1)\}$ containing consecutive vertices with size x and their common representative $R(W) = \cap R^c(u_i)$ with size y, the saving of current window is approximated by $x \times y - x - y$ since common representative $R(W)$ can be replaced with one indexing id for each row. Using the intersection of the compressed transitive closure, an incremental update is enabled as we will see. We consider the current saving as the lower bound of this window, i.e., the saving of sliding window $W = \{u_i, a \leq i \leq (a+x)\}$ containing $x+1$ rows should not decrease compared to the one with x rows. Formally, it requires the following inequality to be satisfied:

$$xy - x - y \leq (x+1)y' - (x+1) - y' \tag{1}$$

where y' is the size of common representative of new window with $x+1$ rows, i.e., $y' = |R(W) \cap R^c(u_{a+x})|$. If this requirement is violated, we simply output the existing window as a new initial group and starts with a new window. Thus, this sliding-window approach also determine the number of groups, which is the number of windows generated by this approach. Clearly, this initialization takes only linear time.

Algorithm 6 Sliding-Window-Initilization(G)

```
1: organize V(G) in the order of reverse topological sort on G;
2: W ← u_1;
3: R(W) ← {u_1};
4: for i = 2 to |V(G)| do
5:    if saving is not decreased if u_i is added based on inequality 1 then
6:       W ← W ∪ {u_i}, R(W) ← R(W) ∩ R^c(u_i);
7:    else
8:       output a new intial group W;
9:       W ← {u_i}, R(W) ← R^c(u_i);
10:   end if
11: end for
12: output a new group in W
```

The sliding-window algorithm is outlined in Algorithm 6. To begin with, we reorganize the vertices with reversed topological sort and initialize the sliding window W using the vertex u_1 (Line 1 to 3). In the main for loop, we apply the inequality condition 1 to determine whether current vertex u_i should be included in window W or not (Line 5). If so, the window and representative are updated simultaneously (Line 6). If the inequality cannot be satisfied, the current window W is output as a new initial group and a new window is created starting from vertex u_i (Line 8 and 9). The procedure is terminated when all vertices are processed. Clearly, each vertex is processed only one time and thus the time complexity of algorithm is linear.

6. EXPERIMENTS

In this section, we empirically evaluate our path-tree cover approach on both synthetic and real datasets. We are particularly interested in the following two key questions:

(1) What are the query time, index size, and construction time of the path-tree approach, compared to state-of-the-art graph indexing schemes, the optimal tree cover approach, 2-hop labeling, 3-hop labeling and GRAIL?
(2) How much benefit we can gain from the chain-tree and post-labeling compression on top of the path-tree indexing approach?

We apply different methods as follows: 1) *PTree-1*, which corresponds to the path-tree approach utilizing optimal tree cover together with *OptIndex* (Subsection 3.1), 2)

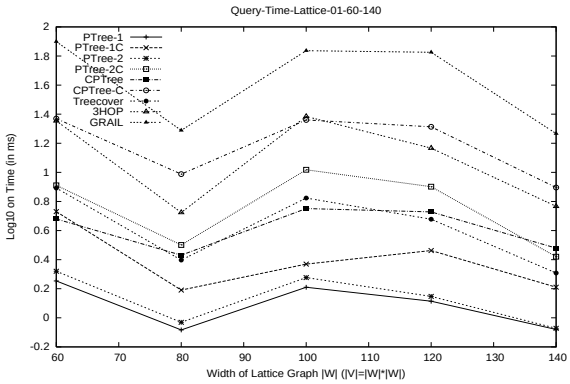


Fig. 9. Query Time for Square-grid Graph

PTree-1C, which corresponds to PTree-1 with post-processing step, 3) *PTree-2*, which corresponds to the path-tree approach described in Section 2 and utilizing the *MinPathIndex* criteria, 4) *PTree-2C*, which corresponds to PTree-2 with post-processing step, 5) *CPTree*, which is chain-tree approach utilizing the same path-decomposition as PTree-1, 6) *CPTree-C*, which is the *CPTree* with post-processing step, 7) *Treecover*, which corresponds to the optimal tree cover approach by Agrawal [Agrawal et al. 1989] and is also referred to as *Interval* in [Wang et al. 2006; Yildirim et al. 2010]. 8) *2HOP*, which corresponds to 2-hop labeling, 9) *3HOP*, which corresponds to 3-hop labeling, and 10) *GRAIL*, which corresponds to the latest scalable reachability indexing approach [Yildirim et al. 2010]. The number of intervals (d) is set to be 5 in GRAIL as being suggested in [Yildirim et al. 2010]. For each method, we measure three parameters: query time, index size, and construction time. To facilitate comparisons among different graph indexing schemes, we measure the index size as the number of integers which are recorded to answer the reachability queries. This measurement is generally consistent with [Wang et al. 2006; Jin et al. 2009; Yildirim et al. 2010]. Furthermore, the construction time is the total processing time of a DAG. As an example, for the path-tree cover approach, it includes the construction of both the path-tree cover and the compressed transitive closure.

All tests were run on an Intel Xeon 3.2GHz machine with 4GB of main memory, running Linux with a 2.6.18 x86_64 kernel. All algorithms are implemented in C++. All PTree-1, PTree-2 and CPTree use an improved version of Algorithm 3 (with $O(mk)$ construction time) to calculate compressed transitive closures. In this implementation we have also fixed some programming bugs in the early version [Jin et al. 2008]. A query is generated by randomly picking a pair of nodes for a reachability test. We measure the query time by answering a total of $100,000$ randomly generated reachability queries.

6.1. Small Synthetic Datasets

In the following, we first investigate different indexing approaches using Grid type of graphs (i.e., each vertex in the graph links to its right and upper corners, and then randomly rewire a small portion of edges) [Schenkel et al. 2004]. Then, we study those approaches using random directed graphs generated from both Erdös-Rényi model and scale-free (power-law degree distribution) on relatively small datasets (with the number of vertices less than $10,000$). In the next subsection, we will study them on large datasets.

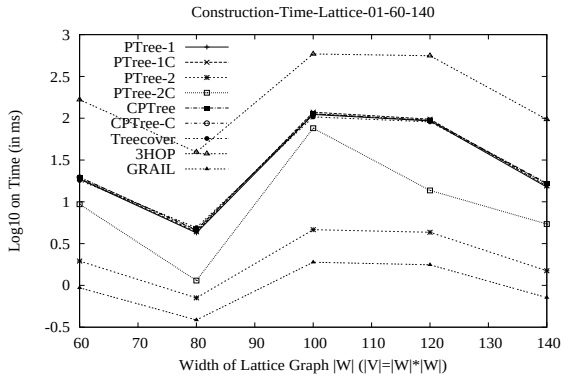


Fig. 10. Construction Time for Square-grid Graph

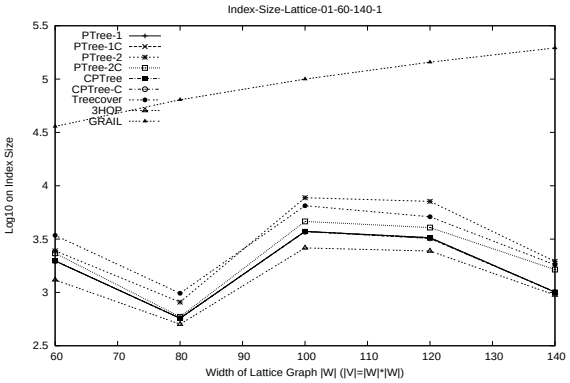


Fig. 11. Index Size for Square-grid Graph

In the first experiment, we generate a set of square-grid graphs with a small portion (1%) of random jumping edges. Specifically, each vertex in the grid (except the boundary ones) has a high probability (99%) linking to its immediate right neighbor (and its immediate upper neighbor) and a small probability (1%) randomly linking to any other vertices in the graph. Figure 9 and Figure 10 show the query time (logarithmic scale) and construction time (logarithmic scale) of the path-tree and chain-tree approaches (PTree-1, PTree-1C, PTree-2, PTree-2C, CPTree and CPTree-C), compared with the optimal tree cover approach (Treecover), 3-hop labeling scheme and GRAIL, respectively with the width of square-grid graphs varying from 60 to 140 (2-hop is omitted in this experiment due to its high construction cost). The index size of these approaches (logarithmic scale) are presented in Figure 11. Clearly, PTree-1 always obtains the best results among path-tree, optimal tree cover and 3-hop in terms of query time. It is on average approximately 3.5 times, 5.2 times and 16.8 times faster than the optimal tree cover, 3-hop and GRAIL, respectively. As shown in Figure 10, PTree-2 is the fastest algorithms on constructing reachability index except GRAIL, and construction cost of 3-hop is most expensive among all algorithms. Moreover, the construction time of PTree-1, PTree-2 and Treecover are comparable to each other. Figure 11 shows that the 3-hop has the smallest index size whereas the GRAIL and optimal tree cover have the largest index size. Especially, PTree-1, PTree-1C, CPTree, and CPTree-C have almost the same index size (their corresponding lines are overlapped in the logarithmic

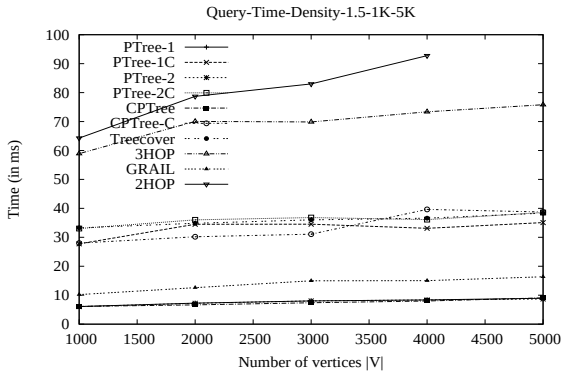


Fig. 12. Query Time for Random DAG with $|E|/|V| = 1.5$

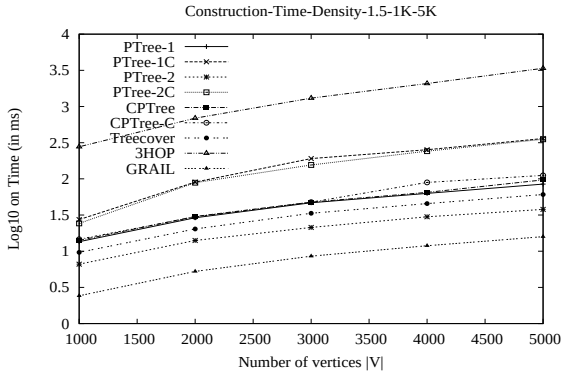


Fig. 13. Construction Time for Random DAG with $|E|/|V| = 1.5$

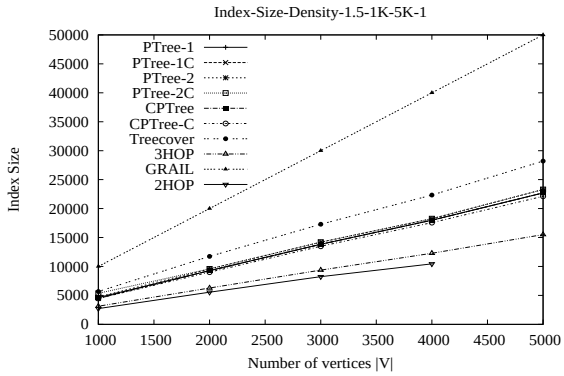


Fig. 14. Index Size for Random DAG with $|E|/|V| = 1.5$

scale figure), and their index sizes are quite close to the one of 3-hop. Overall, the path-tree approach offers the fastest query processing time with the close-to-minimal index size for the Grid type of graphs.

In the second experiment, we generate a set of random DAGs with average edge density (i.e. $|E|/|V|$) of 1.5 and we vary the number of vertices from $1,000$ to $5,000$. For density 1.5, Figure 12 and Figure 13 show the query time and the construction time, respectively, of the path-tree and chain-tree approaches (PTree-1, PTree-1C, PTree-2, PTree-2C, CPTree and CPTree-C), compared with the optimal tree cover approach (Treecover), 2-hop, 3-hop labeling scheme and GRAIL; Figure 14 shows the index size of these approaches, respectively.

For this experiment, we can make the following observations: 1) For the query time, PTree-1, PTree-2, and CPTree are the fastest, and they are on average about 4.7 times faster than that of the optimal tree cover approach, 1.8 times faster than that of the GRAIL approach. The 2-hop approach has the slowest query time and is almost 10.7 times slower than that of PTree-1, PTree-2, and CPTree. In addition, it cannot scale to graphs with $5,000$ vertices due to its high memory and computational cost. The 3-hop approach is about 1.2 times faster than that of the 2-hop and is still almost 9 times slower than that of the fastest path-tree approaches. Furthermore, even the query procedure of the algorithms with post-labeling compression needs an recovering step, the query time of PTree-1C and CPTree-C are still slightly better than that of the optimal tree cover approach. 2) In terms of the construction time, GRAIL is the fastest approach since it only requires a constant number (5) of Depth-First-Search (DFS) traversals of the graphs to generate the reachability indices. Besides that, PTree-2 has the second fastest construction time since it does not rely on the full transitive closure computation, and PTree-1, CPTree, and optimal tree cover approach are comparable. Note that since 2-hop labeling is two orders of magnitude larger than other approaches, we omit their construction time in Figure 13. The construction time of the 3-hop approach is almost 30 times slower than that of the PTree-1 approach. The post-labeling compression also introduces some overhead in terms of construction time but it is still faster than that of the 3-hop and 2-hop approaches. 3) The 2-hop and 3-hop have the smallest index size with 2-hop being slightly better. The index size of PTree-1, PTree-2, and CPTree are comparable, and they are on average approximately 80% of the one from the optimal tree cover, and about 1.5 times larger than that of the 3-hop approach. CPTree and PTree-1 are better than PTree-2 in terms of index size, but CPTree is exactly the same with PTree-1. This in some sense demonstrates the optimality of PTree-1. Though theoretically chain-tree indexing is not worse than path-tree indexing, its practical application seems limited at least based on the current implementation. In particular, what chain decomposition can result in the most compact final transitive-closure is still an open problem (Section 4). Also, the effect of chain-tree optimality given the path-decomposition (Subsection 4.2) seems to be quite small in the tested graphs. We also note that in this experiment, the post-labeling compression step only helps reduce the index size of the path-tree approaches by a very small margin (less than 1%). Further analysis shows that when the graph is small, the redundancy in the compressed transitive closure seems also quite small, and thus the effect of the post-labeling compression is quite limited. As we will show that in Subsection 6.2, the post-labeling compression is more effective in reducing the index size of the large DAGs.

Next, we generate the random graphs with vertex outgoing degree following power-law distribution using a publicly available web graph generator [2]. The edge and vertex ratio is around 2, i.e. $|E|/|V| \approx 2$. Figure 15, Figure 16, and Figure 17 report the query time, construction time, and index size, respectively. For query time, CPTree and PTree-1 are the fastest ones among all algorithms, which are on average around 36 times faster than that of GRAIL, 11 times faster than that of the 2-hop and 3-hop

[2]http://pywebgraph.sourceforge.net/

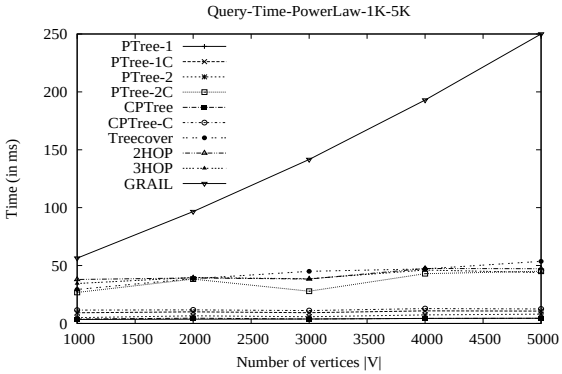


Fig. 15. Query Time for PowerLaw graphs

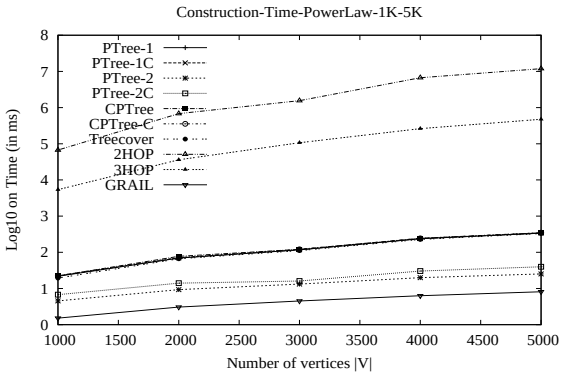


Fig. 16. Construction Time for PowerLaw graphs

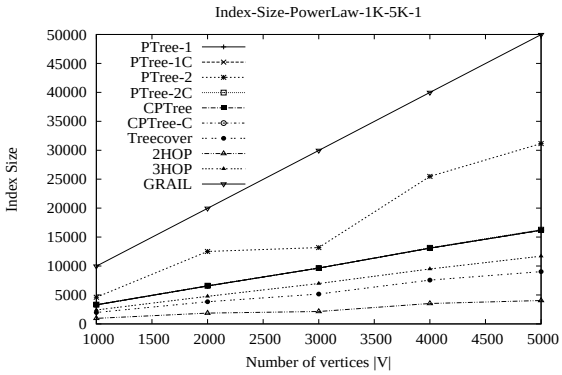


Fig. 17. Index Size for PowerLaw graphs

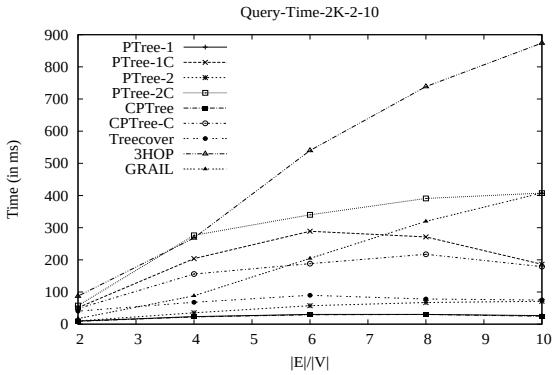


Fig. 18. Query Time for Random DAG with $|V| = 2000$ varying $|E|/|V|$ from 2 to 10

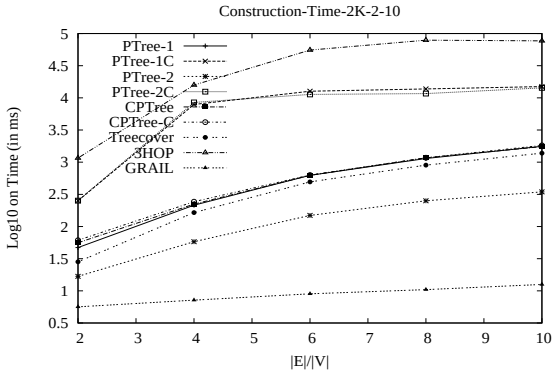


Fig. 19. Construction Time for Random DAG with $|V| = 2000$ varying $|E|/|V|$ from 2 to 10

approaches, and 10 times faster than that of the optimal tree cover approach. As shown in Figure 16, GRAIL has the fastest construction time, and the construction time of 2-hop and 3-hop are at least 2 orders of magnitude slower than the rest of approaches. The PTree-2 and PTree-2C are comparable and PTree-1, PTree-1C, CPTree, CPTree-C, and optimal tree cover have almost the same construction time. We note that the additional post-labeling compression takes little overhead in this experiment. In terms of the index size, the 2-hop, 3-hop, and Treecover achieve the best results among all the approaches. Between them, the index size of 2-hop is around 2 times smaller than the one from optimal tree cover approach. The index size of PTree-1, PTree-1C, PTree-2C, CPTree, and CPTree-C are almost the same, and very comparable to that of the 3-hop. Note that the post-labeling compression step only helps to reduce the index size for PTree-2.

In the forth experiment, we generate random DAGs with $2,000$ vertices, and vary their edge density from 2 to 10. Figure 18 and Figure 19 show the query time and construction time, respectively, of all approaches except 2-hop, which has difficulty in handling dense graphs and therefore is not included in this experiment. Figure 20 shows the index size of all nine approaches. In this experiment, we make the following observations: 1) PTree-1 and CPTree have almost the same query time and they are also the fastest ones. As the edge density increases, the query times of PTree-2 and the optimal tree cover become quite comparable. The query times of the optimal tree cover,

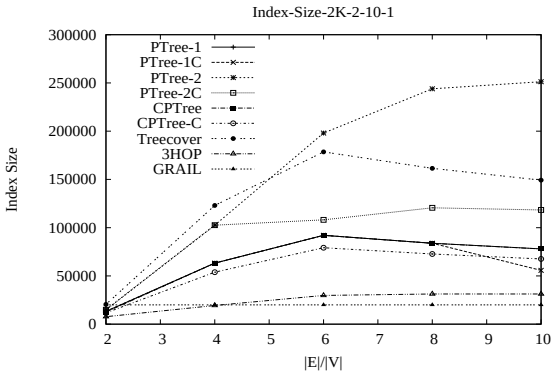


Fig. 20. Index Size for Random DAG with $|V| = 2000$ varying $|E|/|V|$ from 2 to 10

GRAIL, and the 3-hop, are about 3.7 times, 7.6 times, and 19 times, respectively, on average slower than that of the PTree-1 (and CPTree). We also observe that the query times of 3-hop and the GRAIL are affected by the density quite significantly, i.e., they increases when the density increases. 2) For the construction time, the GRAIL is the least expensive one and the 3-hop is the most expensive one. Without the post-labeling compression, the path-tree approach (PTree-1) and optimal tree cover approach are quite comparable and PTree-2 is faster than both of them as expected. In addition, the post-labeling compression introduces some additional costs for PTree-1 and PTree-2. 3) The 3-hop has the smallest index size when the edge density is between 2 and 4; and the GRAIL has the smallest index size when the edge density is no less than 4. Overall, their index sizes are quite comparable. The index sizes of PTree-1 and PTree-2 are approximately 2.6 and 6 times, respectively, larger than that of the 3-hop approach; and they are around 54% of the index size from optimal tree cover approach. 4) The post-labeling compression is shown to work well for the PTree-2 when the edge density is larger than 4: the index size of PTree-2C is approximately half of that in PTree-2. It also brings some benefit to the chain-tree approach CPTree. On average, CPTree-C reduces the index size of CPTree by approximately 13%. However, as shown in Figure 18, such compression also increases the query time: the query time of PTree-2C is on average 6.2 times slower than that of the PTree-2.

To sum, these experiments demonstrate (for relatively small directed graphs, $|V| \leq 10,000$): 1) for the sparse graph ($|E/|V| \leq 2$), the PTree-1 seems to be the best approach in terms of index size (is comparable with 3-hop) and query time. 2) the improvement from the chain-tree indexing (CPTree) seems to be limited, but the post-labeling compression show some improvements (up to 50% index size reduction). 3) the 3-hop indexing tends to have the smallest index size (or close to) but with expensive construction time. In addition, its query time is slower than path-tree based approaches. 4) The GRAIL has the fastest construction time. Its index size is also very small. However, its query time is at least one order of magnitude slower than that of PTree-1 on the dense random graphs and power-law graphs.

6.2. Large Synthetic Datasets

In the following experiments, we study the performance of our algorithms on relatively large synthetic graphs. Since both 2-hop and 3-hop cannot scale to very large graph because of their high memory cost and very long construction time for 2-hop, we exclude both of them in comparison. In addition, the chain-tree approach is also omitted due to its almost equivalent performance compared with PTree-1. Thus, we focus on compar-

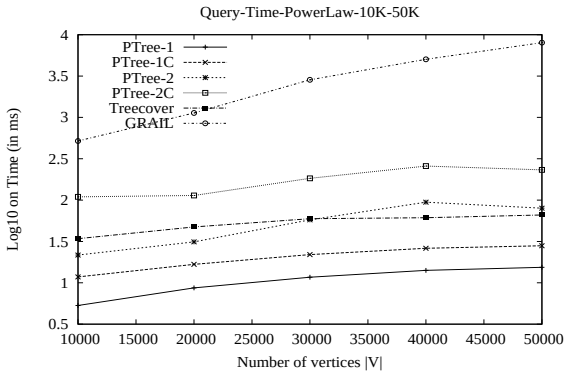


Fig. 21. Query Time for large power-law graphs

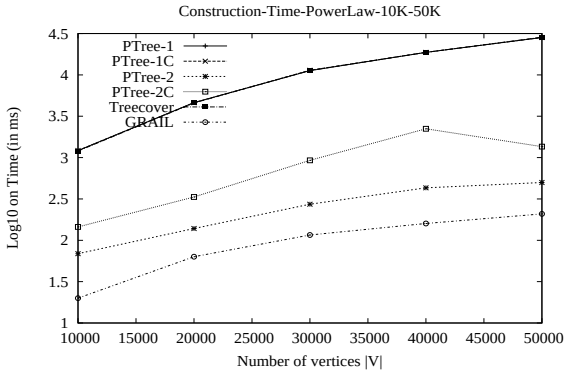


Fig. 22. Construction Time for large power-law graphs

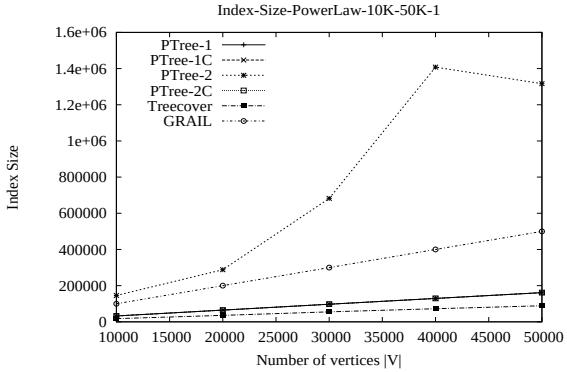


Fig. 23. Index Size for large power-law graphs

ing the algorithms PTree-1, PTree-1C, PTree-2, PTree-2C with Treecover and GRAIL here.

In the first experiment, we study the indexing performance on the random graphs with the power-law degree distribution. We vary the number of vertices from $10,000$

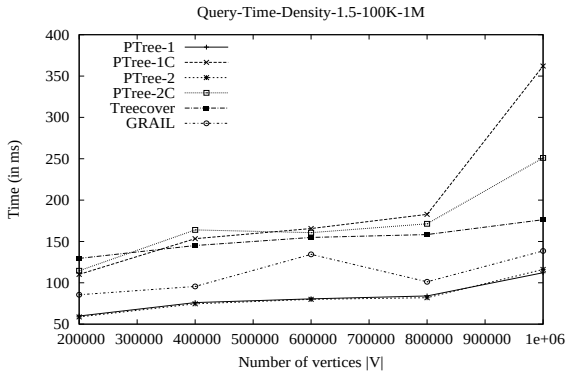


Fig. 24. Query Time for Large Random DAG with $|E|/|V| = 1.5$

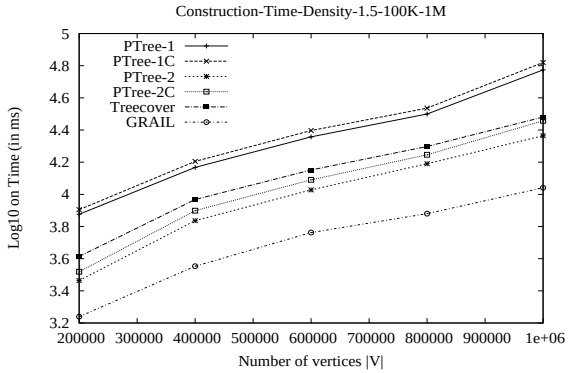


Fig. 25. Construction Time for Large Random DAG with $|E|/|V| = 1.5$

to $50,000$ while fixing the edge density as 1.5. Figure 21 and Figure 22 illustrates the query time and construction time of our four algorithms, optimal tree cover approach and GRAIL. Figure 23 reports the index size of those approaches. For the query time, PTree-1 and PTree-1C are faster than Treecover on average by 5 times and 2.6 times, respectively. Furthermore, PTree-1, PTree-1C, PTree-2 and PTree-2C are on average approximately 270 times, 140 times, 53 times and 17 times faster than that of GRAIL, respectively. In terms of the construction time, PTree-2 and PTree-2C are much faster than PTree-1, PTree-1C, and TreeCover. Again, GRAIL is always the fast algorithm for constructing reachability indices. The index size of optimal tree cover approach is around half of the one from PTree-1 and PTree-1C, which is also the best results among all algorithms. PTree-2C is significantly better than PTree-2, and on average obtains even 7 times smaller index size than that of PTree-2. Interestingly, even the index size of PTree-1 is much smaller than PTree-2, the post-labeling compression dramatically reduces this gap, such that the index size of PTree2-C is very close to the one of PTree-1C. Finally, the index size of GRAIL is 3 times larger than that of the PTree-1.

In the second experiment, we generate a set of random DAGs with average edge density (i.e. $|E|/|V|$) of 1.5, and we vary the number of vertices from $200K$ to 1 million. The query time and the construction time of our algorithms, optimal tree cover approach and GRAIL are shown in Figure 24 and Figure 25, respectively. Figure 26 reports the index size. Overall, the query time of PTree-1 is approximately 1.9 times

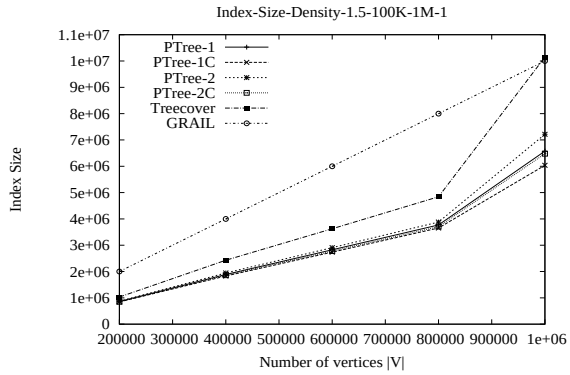


Fig. 26. Index Size for Large Random DAG with $|E|/|V| = 1.5$

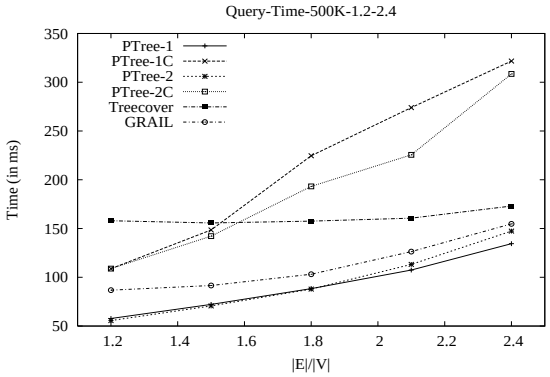


Fig. 27. Query Time for Random DAG with $|V| = 500K$ varying $|E|/|V|$ from 1.2 to 2.4

and 1.4 times faster than that of optimal tree cover approach and GRAIL. Even with additional operations needed in PTree-1C, its query time is still very close to the optimal tree cover approach. In terms of the construction time, GRAIL is the fastest one. The construction time of PTree-1 is around 2 times slower than that of PTree-2, which is also faster than that of the optimal tree approach. The post-labeling compression (PTree-1C and PTree-2C) introduces around 10% extra construction time with respect to the original path-tree approaches. For the index size, PTree-1, PTree-1C, PTree-2 and PTree-2C are approximately 76.4%, 73.5%, 79% and 75.2%, respectively, of the one from the optimal tree cover approach. The index size of GRAIL is the largest one and it is approximately 2 times larger than that of PTree-1.

In this experiment, we generate random DAGs with $500,000$ vertices, and vary their edge density from 1.2 to 2.4. Figure 27, 28 and 29 show the query time, construction time, and index size of five algorithms. The query time of PTree-1 and PTree-2 are on average approximately 81% and 82% of the one from GRAIL. They are also on average about 1.9 and 1.8 times faster than Treecover, respectively. In terms of construction time, GRAIL is clearly the winner of all algorithms. PTree-2 is faster than the rest of algorithms except GRAIL, and PTree-2C is comparable with Treecover. Overall, the index size of PTree-1, PTree-1C, PTree-2 and PTree-2C are approximately 63%, 60%, 68% and 64%, respectively, of the one from Treecover. Since the number of vertices is fixed in this experiment, the index size of GRAIL is constant.

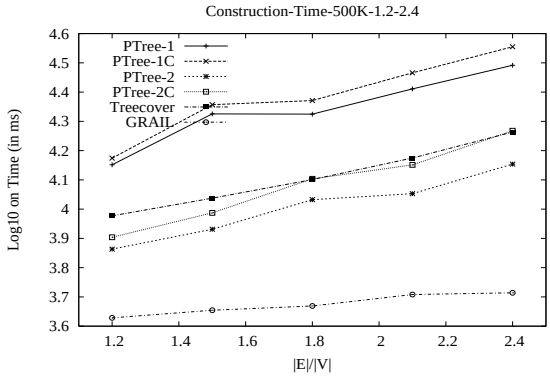


Fig. 28. Construction Time for Random DAG with $|V| = 500K$ varying $|E|/|V|$ from 1.2 to 2.4

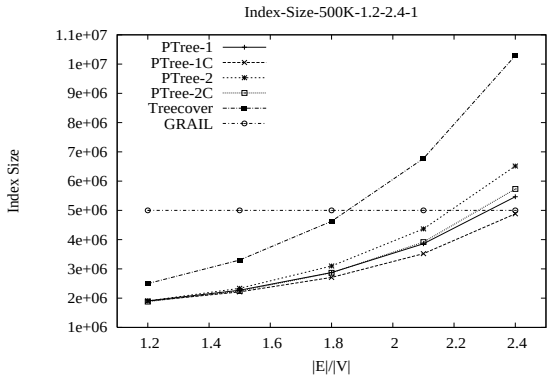


Fig. 29. Index Size for Random DAG with $|V| = 500K$ varying $|E|/|V|$ from 1.2 to 2.4

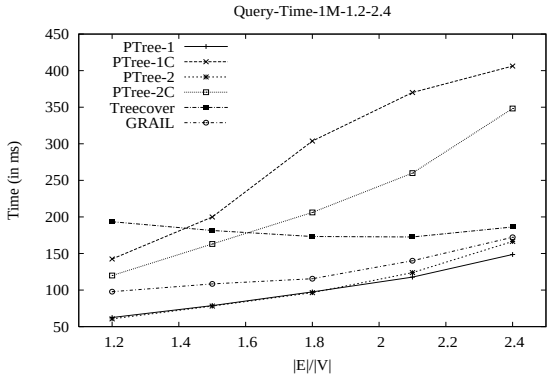


Fig. 30. Query Time for Random DAG with $|V| = 1M$ varying $|E|/|V|$ from 1.2 to 2.4

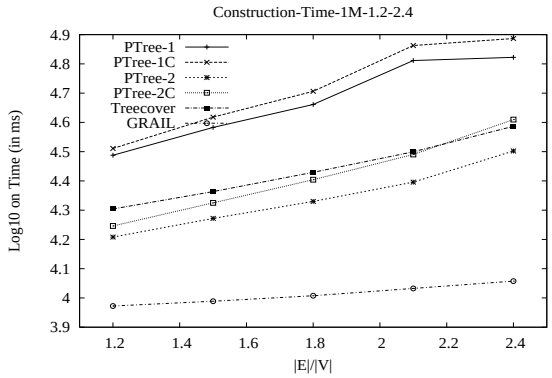


Fig. 31. Construction Time for Random DAG with $|V| = 1M$ varying $|E|/|V|$ from 1.2 to 2.4

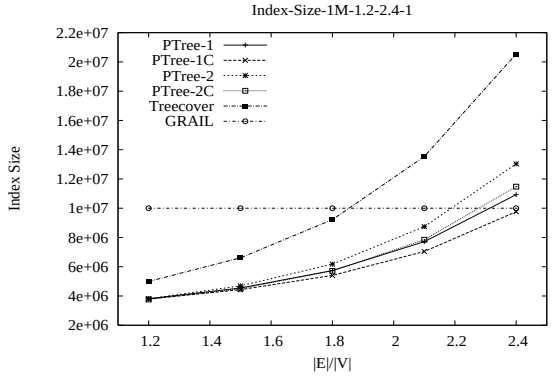


Fig. 32. Index Size for Random DAG with $|V| = 1M$ varying $|E|/|V|$ from 1.2 to 2.4

In the last experiment for synthetic data, we study the performance on random DAGs with $1M$ vertices, varying their edge density from 1.2 to 2.4. Figure 30 and 31 present the query time and construction time of our algorithms, the optimal tree cover approach and GRAIL. Figure 32 reports the index size of all six algorithms. The overall observation is consistent with the last experiment for the random DAGs with $500K$ vertices. In terms of query time, both PTree-1 and PTree-2 are on average approximately 2 times faster than Treecover. In addition, PTree-1 and PTree-2 are faster than GRAIL by an average of 22% and 20% for all graphs. For construction time, GRAIL still obtain the best performance. PTree-2 is much faster than the rest of algorithms, and Treecover is comparable with Ptree-2C. Overall, the index size of our algorithms PTree-1, PTree-1C, PTree-2 and PTree-2C are approximately 64%, 60%, 68% and 64%, of the one from optimal tree cover approach, respectively.

6.3. Real Datasets

In this section, we study different approaches in answering reachability queries on a list of real datasets (listed in Table II). Among them, Anthra, Ecoo157, Mtbrv and VchoCyc are from EcoCyc [3]; Xmark and Nasa are XML documents; and KEGG is metabolic networks used in [Trißl and Leser 2007]. The last three with relatively large

[3] `http://ecocyc.org/`

Table II. Real Datasets

Graph Name	#V	#E	DAG #V	DAG #E
Anthra	13736	17307	12499	13104
Ecoo157	13800	17308	12620	13350
Kegg	14271	35170	3617	3908
Mtbrv	10697	13922	9602	10245
Nasa	5704	7942	5605	7735
Vchocyc	10694	14207	9491	10143
Xmark	6483	7654	6080	7028
Arxiv	6000	66707	6000	66707
Go	6793	13361	6793	13361
Yago	6642	42392	6642	42392

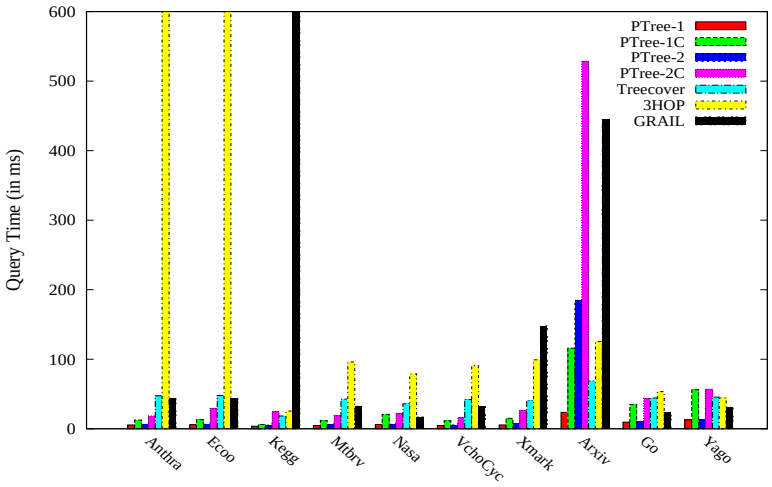


Fig. 33. Query Time for Real Datasets

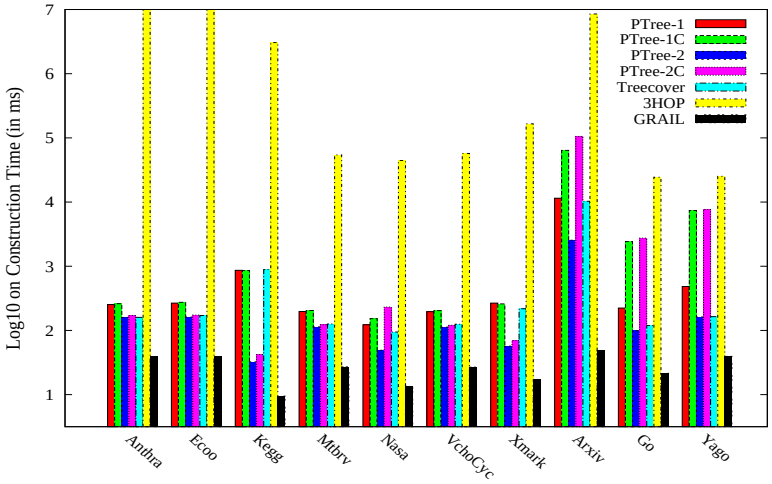


Fig. 34. Construction Time for Real Datasets

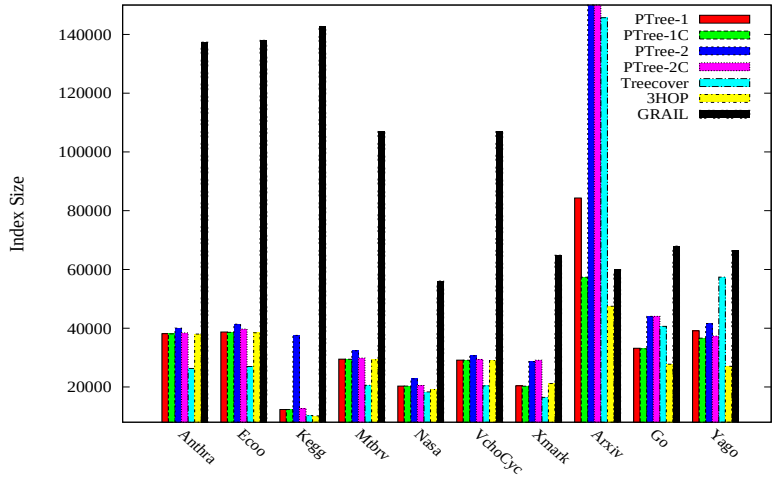


Fig. 35. Index Size for Real Datasets

density are directly extracted from the real-world large datasets with density being larger than or close to 2. The extraction would directly produce DAGs, and therefore we have $(\#V = DAG\#V)$ and $(\#E = DAG\#E)$. Specifically, ArXiv is extracted from a dataset of citations among scientific papers from the arxiv.org website [4]. GO contains genetic terms and their relationships from the Gene Ontology project [5]. Yago describes the structure of relationships among terms in the semantic knowledge database from the YAGO [6]. The first two columns in Table II are the number of vertices and edges in the original graphs, and the last two columns are the number of vertices and edges in the DAG after compressing the strongly connected components.

Figure 33 and Figure 34 show the query time and construction time of our path-tree, 3-hop and GRAIL. Figure 35 illustrates the index size of those approaches. For the query time, we can see that PTree-1 consistently achieves the fastest query time and PTree-2 is quite comparable to PTree-1 on almost all datasets (except Arxiv). Note that for display purpose, Figure 33 only shows the query time up to 600 milliseconds. The query times of 3-hop are about $1,400$ms for both Anthra and Ecoo; and the query time of GRAIL is about 700ms for Kegg. Overall, PTree-1 is on average approximately 58 times and 27 times faster than that of 3-hop and GRAIL. The query time of the optimal tree cover approach is slower than that of the PTree-1C and PTree-2C on the first seven sparse datasets and is faster (or quite comparable) on the three relatively dense datasets. For the construction time, as we expected, GRAIL is the fast algorithm since only a constant number of DFS traversals is needed, and 3-hop is the most expensive one among all algorithms. On average, 3-hop is on average around 3 and 4 orders of magnitude slower than PTree-1 and GRAIL, respectively. We also observe that PTree-1 and PTree-1C are slower than the optimal tree cover approach since it uses the optimal tree cover as the first step for path-decomposition (Recall that we extract the paths from the optimal tree). However, PTree-2 uses less construction time than optimal tree cover in all datasets, and on average is 3.8 times as fast as the optimal tree cover. This result is generally consistent with our analysis of the theoretical time complexity,

[4] http://arxiv.org/
[5] http://www.geneontology.org/
[6] http://www.mpi-inf.mpg.de/suchanek/downloads/yago/

which is $O(m+n\log n)+O(mk)$. Even PTree-2C is on average 2.7 times faster than that of the optimal tree cover with respect to construction time. In terms of the index size, the optimal tree cover tends to the smallest for the sparse graphs and the 3-hop obtains the best results for the dense graphs. In addition, on the seven sparse graphs, PTree-1 and 3-hop are quite comparable. On most of the datasets, PTree-2 is only slightly larger than PTree-1. However, on Arxiv, it is 14 times worst than that of the PTree-1, which also explains why it is much slower than PTree-1 on this dataset. Again, for display purpose, Figure 35 only shows the index size up to $150K$. The index sizes of PTree-2 and PTree-2C on Arxiv are about $1187K$ and $202K$, respectively. In addition, on the dense datasets, the post-labeling compression seems offering reasonable reduction for the index size though there is little improvement on the sparse datasets.

6.4. Summary

To sum, the experimental results on both synthetic and real datasets demonstrates that the path-tree indexing approaches work well on Grid-type of graphs, small DAGs ($|V| \leq 10,000$), large sparse graphs ($|E|/|V| \leq 2$), and the existing real benchmark graphs in terms of both query time and index size in most cases. They are also very easy to build. The post-labeling compression help further reduce the index size of the path-tree approaches especially on dense graphs. The 3-hop is the best in reducing index size in dense graphs but its query time is slower than the path-tree methods. In addition, it is very expensive to build. The GRAIL approach is the fastest algorithm in constructing reachability indices and its index size is generally moderate (determined by the number of vertices and the number of intervals used in the index). However, it seems having difficulty in handling scale-free graphs and dense graphs (its query time can be up to two orders of magnitude slower than that of the path-tree methods on these graphs). Finally, the path-tree methods can handle a graph with up to one million vertices on a 4GB machine.

7. CONCLUSION

In this paper, we introduce a novel *path-tree* structure to assist with the compression of transitive closure and answering reachability queries. In addition, we study several improvements of the path-tree, including its generalization, chain-tree, and additional compression based on the "reachability similarity". Our experimental evaluation demonstrates the path-tree approaches have the fastest query answering time and comparable index size compared with the state-of-art indexing construction techniques, including optimal tree cover, 2-hop, 3-hop, and GRAIL. It is also easy to build. In the future, we will investigate how to construct disk-based path-tree approaches and how construct path-tree in parallel computer (especially the Cloud environment) for reachability queries.

REFERENCES

ADLER, M. AND MITZENMACHER, M. 2002. Towards compressing web graphs. In *Data Compression Conference, 2001. Proceedings. DCC 2001.* IEEE, 203–212.

AGRAWAL, R., BORGIDA, A., AND JAGADISH, H. 1989. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. ACM, 253–262.

BOUROS, P., SKIADOPOULOS, S., DALAMAGAS, T., SACHARIDIS, D., AND SELLIS, T. K. 2009. Evaluating reachability queries over path collections. In *SSDBM*. 398–416.

CHEN, L., GUPTA, A., AND KURUL, M. 2005. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 493–504.

CHENG, J., YU, J. X., LIN, X., WANG, H., AND YU, P. S. 2006. Fast computation of reachability labeling for large graphs. In *EDBT*. 961–979.

CHU, Y. J. AND LIU, T. H. 1965. On the shortest arborescence of a directed graph. *Science Sinica 14*, 1396–1400.

COHEN, E., HALPERIN, E., KAPLAN, H., AND ZWICK, U. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput. 32,* 5, 1338–1355.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*. The MIT Press.

DE BERG, M., CHEONG, O., VAN KREVELD, M., AND OVERMARS, M. 2008. *Computational Geometry: Algorithms and Applications* Third Ed. Springer-Verlag.

DILWORTH, R. P. 1950. A decomposition theorem for partially ordered sets. *The Annals of Mathematics, Second Series 51,* 1, 161–166.

EDMONDS, J. 1967. Optimum branchings. *J. Research of the National Bureau of Standards 71B*, 233–240.

GABOW, H. N., GALIL, Z., SPENCER, T., AND TARJAN, R. E. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica 6,* 2, 109–122.

GOLDBERG, A. V., TARDOS, E., AND TARJAN, R. E. 1990. *Network Flow Algorithms*. Springer Verlag, 101–164.

JAGADISH, H. V. 1990. A compression technique to materialize transitive closure. *ACM Trans. Database Syst. 15,* 4, 558–598.

JIN, R., HONG, H., WANG, H., RUAN, N., AND XIANG, Y. 2010. Computing label-constraint reachability in graph databases. In *SIGMOD Conference*. 123–134.

JIN, R., XIANG, Y., RUAN, N., AND FUHRY, D. 2009. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD Conference*. 813–826.

JIN, R., XIANG, Y., RUAN, N., AND WANG, H. 2008. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD Conference*. 595–608.

KAMEDA, T. 1975. On the vector representation of the reachability in planar directed graphs* 1. *Information Processing Letters 3,* 3, 75–77.

KÖNIG, J. 1884. Über eine eigenschaft der potenzreihen. *Math. Ann. 23*, 447–449.

NAVLAKHA, S., RASTOGI, R., AND SHRIVASTAVA, N. 2008. Graph summarization with bounded error. In *SIGMOD Conference*. 419–432.

RAGHAVAN, S. AND GARCIA-MOLINA, H. 2003. Representing web graphs. In *ICDE*. 405–416.

SCHENKEL, R., THEOBALD, A., AND WEIKUM, G. 2004. Hopi: An efficient connection index for complex xml document collections. In *EDBT*. 237–255.

SIMON, K. 1988. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci. 58,* 1-3, 325–346.

TRISSL, S. AND LESER, U. 2007. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*. 845–856.

WANG, H., HE, H., YANG, J., YU, P. S., AND YU, J. X. 2006. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*. 75.

YILDIRIM, H., CHAOJI, V., AND ZAKI, M. J. 2010. Grail: scalable reachability index for large graphs. *Proc. VLDB Endow. 3*, 276–284.

A. PROOF OF LEMMA 5

To prove this lemma, we first prove the claim:

$$v \in R^c(u) \Longleftrightarrow u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$$

For vertex v, if $v \in R^c(u)$, then $u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$. This can be proved by contradiction. Assume $u \notin S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$. It's easy to see $\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}) \subseteq S(v)$. Hence there are two cases. (case 1:) $u \in \bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\})$. Then there must exist a vertex w such that u can reach w in DAG G and w can reach v in the path tree. Then v should be replaced by w in $R^c(u)$, a contradiction. (case 2:) $u \notin S(v)$. Then we get $v \notin R^c(u)$, a contradiction.

For vertex u, if $u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$, then $v \in R^c(u)$. This can also be proved by contradiction. Assume $v \notin R^c(u)$. Then because $u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$ and $\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}) \subseteq S(v)$, which implies $u \in S(v)$ and $u \notin \bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\})$, we conclude u cannot reach v (i.e. $u \notin S(v)$) if $v \notin R^c(u)$, a contradiction.

Thus, for each vertex

$$v \in R^c(u) \Longleftrightarrow u \in S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))$$

Then it's easy to see

$$Index\_cost = \sum_{u \in V(G)} R^c(u) = \sum_{v \in V(G)} |S(v) \backslash (\bigcup_{x \in T_{pre}(v)} (S(x) \cup \{x\}))|$$

□

B. PROOF OF LEMMA 6

Proof: The key property is that for any two vertices w and w' in $\mathcal{I}_{mid}$, either $w.I \subseteq w'.I$ or $w'.I \subseteq w.I$. This is because the intervals are extracted from the tree structure, and it is easy to see that:

$$w.I \cap w'.I \neq \emptyset \Rightarrow (w.I \subseteq w'.I) \vee (w'.I \subseteq w.I)$$

Thus, we can order the intervals based on the inclusion relationship. Further, if $w.I \subseteq w'.I$, then we have $w.X < w'.X$.

This is because if $w.X > w'.X$ (noting that X label is unique) and $w.I \subseteq w'.I$, which means w' can reach w by lemma 3 and theorem 1, w should be dropped from $R(u)$ by algorithm 3 for maximum compression of transitive closure.

Following this, we can see that for any w from Line 2 if $w.I$ includes $v.I$, we can tell w can reach v by lemma 3 and theorem 1, and thus u can reach v. If $w.I$ does not include $v.I$, then no other vertex in $\mathcal{I}_{mid}$ can include $v.I$ with $w.X \leq v.X$, and thus u cannot reach v. □

C. PROOF OF LEMMA 7

Proof: It is easy to see that each edge in $E^R_{P_j \to P_i}$ crosses (see Definition 3) an edge in $E'^R_{P_j \to P_i}$. Therefore, to prove this lemma, we only need to prove the following claim:

Let $\{(u_1, v_1), (u_2, v_2), \ldots, (u_q, v_q)\}$ be the set of edges in $E^R_{P_j \to P_i}$ that cross the edge (u, v) in $E'^R_{P_j \to P_i}$. Then we claim that $\sum_{i=1}^{q} |S_{u_i}(v_i)| \leq |S_u(v)|$.

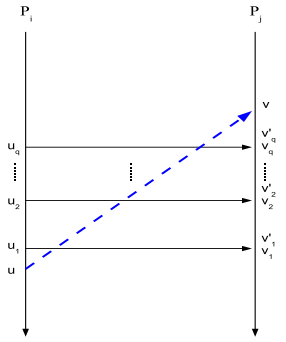


Fig. 36. Illustration of proof of Lemma 7

Given the edge (u, v) in $E'^{R}_{P_j \to P_i}$ (See Figure 36 as an illustration), it is easy to see that $S(u_i) \cup \{u_i\} \subseteq S(v'_i) \cup \{v'_i\}$ for $i = 1$ to $q - 1$. According to the definition of $S_u(v)$ in Section 3.1 that $S_u(v) = (S(u) \cup \{u\}) \backslash (S(v') \cup \{v'\})$, we conclude that $S_{u_i}(v_i) = \emptyset$ for $i = 1$ to $q - 1$. Therefore, $\sum_{i=1}^{q} |S_{u_i}(v_i)| = |S_{u_q}(v_q)| = (S(u_q) \cup \{u_q\}) \backslash (S(v'_q) \cup \{v'_q\}) \leq |S_u(v)|$. □

D. INCREMENTAL UPDATES

It's not necessary to reconstruct path-tree cover when there is an incremental change of G. In this section, we discuss how we incrementally update path-tree cover.

For simplicity of discussion, we define an incremental change to be an addition of an isolated vertex or a deletion of an isolated vertex, or an addition of a new edge or a deletion of an old edge. It's easy to see any change can be decomposed into a series of incremental changes.

It is easy to handle the case of adding or deleting an isolated vertex. When adding a new vertex, we create a new path with containing only the new vertex being added. Then we assign a new $\mathcal{SP}$-tree interval to this new path and a new X label to the new vertex, by connecting this path to the virtual root in the $\mathcal{SP}$-tree. When deleting an isolated vertex, we first delete it from G. If there's a path containing only this vertex, we will delete the path and its incident edge (to the virtual root) from path-graph and $\mathcal{SP}$-tree.

In the following discussion, we will focus on the incremental change of adding a new edge or deleting an old edge. We assume DAG G is the current graph and G' is the graph after applying an incremental changes on G.

Adding an edge in the original directed graph can be classified as either adding an edge within a strongly connected components, which results in no updates, or adding an edge between two strongly connected components (a single vertex can be regarded as a strongly connected component), which can be handled as adding an edge in DAG G.

Similarly, deleting an edge in the original directed graph can be classified as either deleting an edge within a strongly connected components, or deleting an edge between two strongly connected components. The latter case can be handled as deleting an edge in DAG G. For the former case, if deleting the edge does not result in the break of strongly connected component, there is no update. Otherwise, we will have to rerun the path-tree construction from the very beginning, starting with strongly connected component contraction.

D.1. Addition of a new edge

Addition of an edge that creates a strongly connected component If u can reach v in G, adding an edge vu creates a strongly connected component. In this case, we will have to rerun the path-tree construction from the very beginning, starting with strongly connected component contraction.

Addition of a redundant edge

A new edge $(u, v) \in E(G')$ is redundant if u can reach v in G. In this case, adding edge (u, v) to G will result in no update.

Addition of a path-tree edge

A new edge (u, v) is considered as a path-tree edge if and only if:

— (1) It is not a redundant edge.
— (2) (P_u, P_v) is an edge in $\mathcal{S}P$-tree where $u \in P_u$ and $v \in P_v$.

In this case, we add edge (u, v) into the path-tree cover and clear redundant edge from P_u to P_v by calling Algorithm 1. Then we update labeling by Algorithm 7.

Algorithm 7 DFSLabelUpdateAddition($G[P](V, E), P_1 \cup \cdots \cup P_k, (u, v)$)

Parameter: $P_1 \cup \cdots \cup P_k$ is the path-decomposition of G
Parameter: $G[P]$ is represented as linked lists: $\forall v \in V : linkedlist(v)$ records all the immediate neighbors of v. Let $v \in P_i$. *If v is not the last vertex in path P_i, the first vertex in the linked list is the next vertex of v in the path*

```
1: N ← |V|
2: DFS(v)
```

Procedure DFS(v)

```
1: for each v' ∈ linkedlist(v){Assume v' ∈ P_i} do
2:    if v' has no incoming path-tree edge from another path P_j then
3:       DFS(v')
4:    end if
5: end for
6: X(v) ← Label(u).append(N) {Label(u) is u's X label}
7: N ← N − 1
```

Figure 37 shows a running example: Figure 37(a) shows the original path-tree and its labeling. when adding edge $(10, 3)$, which is considered as a path-tree edge, we first clear redundant edge from P_2 to P_1 by calling Algorithm 1 and edge $(7, 6)$ is removed. Then we update labeling by Algorithm 7 and vertex 3, 6, 13 is re-labeled as 11.13, 11.14, and 11.15 respectively. Here and in Section D.2 we assume each label is a *vector* in which numbers are separated by dot(s). When comparing the X labels of two vertices, we follow the *lexicographic order*. For example, $8 < 11 < 11.13 < 11.14 < 11.15 < 12$. To restrict the size of each vector, we will completely re-label each vertices by Algorithm 2 when a vector exceeds certain limit.

We can also use floating-point numbers as an alternative to vectors for the above labeling purpose. For example, in the above case the vertex 3, 6, 13 might be replaced by floating point numbers 11.25, 11.5, and 11.75 respectively. The labeling algorithm are basically the same as Algorithm 7 and we omit it. When a floating-point number overflows, the re-label algorithm, i.e. Algorithm 2, will be triggered.

The correctness of Algorithm 7 is stated in the following lemma.

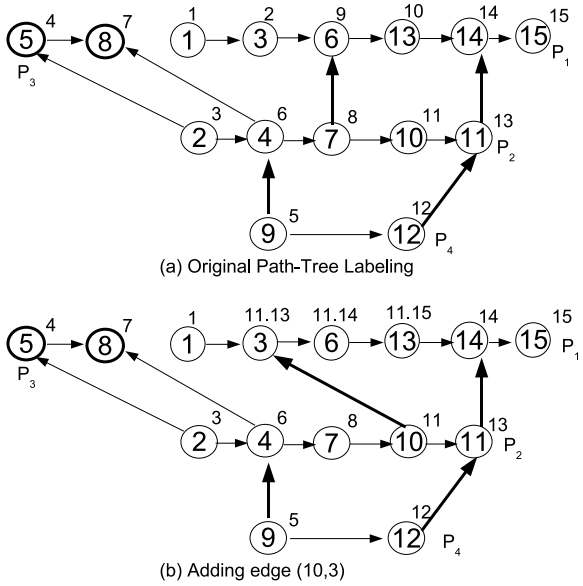


Fig. 37. An example of adding a path-tree edge

LEMMA 8. *Algorithm 7 update the path-tree $G[T]$ to a new path-tree $G'[T']$, where G' has an additional edge uv over G. That is, given two vertices a and b in G', a can reach b through $G'[T']$ if and only if 1) $a.X \leq b.X$ 2) $b.I \subseteq a.I$.*

Proof: To prove this lemma, we first divide the vertices of $G'[T']$ into two disjoint sets: A, in which vertices have not been re-labeled, and B, in which vertices have been re-labeled. Note that only X labels of vertices in A have been re-labeled. The I label of any vertex in $G'[T']$ remains the same.

Second, it is easy to see the lemma is correct if the following two claims hold:
Given two vertices $a \in A$, $b \in B$,
Claim(1): a can reach b through $G'[T']$ if and only if 1) $a.X \leq b.X$ 2) $b.I \subseteq a.I$.
Claim(2): b can reach a through $G'[T']$ if and only if 1) $b.X \leq a.X$ 2) $a.I \subseteq b.I$.

We prove these claims one by one as follows:
Proof of Claim(1):
$\Rightarrow$:

There are two subcases:
Subcase(1) a can reach u through $G'[T']$: By Lemma 4, we have $a.X \leq u.X$ and $u.I \subseteq a.I$. Since I labels have not been changed by Algorithm 7, we have $b.I \subseteq u.I$. According to the re-labeling of Algorithm 7, we have $u.X \leq b.X$. Hence, we conclude that $a.X \leq b.X$ and $b.I \subseteq a.I$, which implies a can reach b through $G'[T']$.
Subcase(2) a cannot reach u through $G'[T']$: In this case, it is not difficult to see $b.I \subseteq a.I \subseteq u.I$ must hold. Thus we conclude that $a.X \leq u.X \leq b.X$. Otherwise, we will find a contradiction that there exist a strongly connected components $a \leftrightarrow b$.
$\Leftarrow$:
If a can reach u through $G'[T']$, then we immediately conclude that a can reach b through $G'[T']$. Let us consider the case a cannot reach u through $G'[T']$. Since $b.I \subseteq a.I$, we conclude that there is a vertex $c \in B$ on the same path of a, and c can reach b through $G'[T']$. Since a and c are on the same path, either a can reach c or c can reach

a through $G'[T']$. If the former case holds, we are done. If the latter case holds, we have b reach a and a reach c through $G'[T']$, a contradiction to the fact that $a \notin B$.
Proof of Claim(2):
$\Rightarrow$:

Since b can reach a through $G'[T']$, it is easy to see that $a.I \subseteq b.I$. Since $a \notin B$, we conclude that there exists a vertex c on the same path of u such that $u.X \leq c.X$. According to the re-labeling of Algorithm 7, we have $b.X \leq c.X \leq a.X$.
$\Leftarrow$:

Since $a.I \subseteq b.I$ and $a \notin B$, we conclude that there exists a vertex $c \in A$ on the same path of b such that c can reach a through $G'[T']$. Since b and c are on the same path, either b can reach c or c can reach b through $G'[T']$. If the former case holds, we are done. If the latter case holds, we have c reach b and b reach a through $G'[T']$, a contradiction to the fact that $b \notin A$.

□

Finally, we need to partially update the transitive closure by calling Algorithm 3 with j being the topological order of v. Or we can update u's transitive closure by merging $v's$ transitive closure (To minimize u's transitive closure, the merging rule will be similar as the adding rule in section 2) and call Algorithm 3 with j being the topological order of u.

Addition of a non-path-tree edge
Addition of a non-path-tree edge is fair easy to handle. Suppose the non-path-tree edge to be added is (u, v). Then we will broadcast v's transitive closure to u and all u's ancestors in reverse topological order, the same as we did previously for adding a path-tree edge. There's no change on path-tree cover and labeling.

D.2. Deletion of an old edge

Deletion of a redundant edge
An edge $(u, v) \in E(G)$ is redundant if u can reach v in G', where $V(G') = V(G)$ and $E(G') = E(G) \backslash \{(u, v)\}$. In this case, deleting edge (u, v) from G will result in no update. However, it is difficult to tell if an edge is redundant without computing the index on G'. Alternatively, we consider two cases, deletion of a path-tree edge and deletion of a non-path-tree edge.

Deletion of a path-tree edge

Assume the path-tree edge to be delete is (u, v). If (u, v) is a path edge, deleting (u, v) will result in path broken and we have to rerun the path-tree construction from the beginning.

If (u, v) is an edge between two paths, deletion of (u, v) will result in deleting it from the path-tree cover and re-labeling some vertices. Algorithm 8 deletes a path-tree edge.

Figure 38 shows a running example: When deleting a path-tree edge $(7, 6)$, we delete it from both G and the path-tree cover. Then we call Algorithm 8 and update the X label (vector) of vertex 6 and 13 by 2.14 and 2.15.

In addition, we need to recompute the transitive closure starting from u by calling Algorithm 3 with j being the topological order of u.

The correctness of Algorithm 8 is stated in the following lemma.

LEMMA 9. *Algorithm 8 update the path-tree $G[T]$ to a new path-tree $G'[T']$, where G' has an additional edge uv over G. That is, given two vertices a and b in G', a can reach b through $G'[T']$ if and only if 1) $a.X \leq b.X$ 2) $b.I \subseteq a.I$.*

Algorithm 8 DFSLabelUpdateDeletion($G[P](V, E), P_1 \cup \cdots \cup P_k, (u, v)$)

Parameter: $P_1 \cup \cdots \cup P_k$ is the path-decomposition of G
Parameter: $G[P]$ is represented as linked lists: $\forall v \in V : linkedlist(v)$ records all the immediate neighbors of v. Let $v \in P_i$. *If v is not the last vertex in path P_i, the first vertex in the linked list is the next vertex of v in the path*

1: $N \leftarrow |V|$
2: **if** v is not the first (root) vertex in its path **then**
3: $L = X$ label of v's previous vertex in its path
4: **else**
5: $L = X$ label of the first (root) vertex in u's path minus 1
6: **end if**
7: DFS(v)

Procedure DFS(v)

1: **for each** $v' \in linkedlist(v)$ {Assume $v' \in P_i$} **do**
2: **if** v' has no incoming path-tree edge from another path P_j **then**
3: DFS(v')
4: **end if**
5: **end for**
6: $X(v) \leftarrow L.append(N)$ {Update vertex v's Label with $L.append(N)$}
7: $N \leftarrow N - 1$

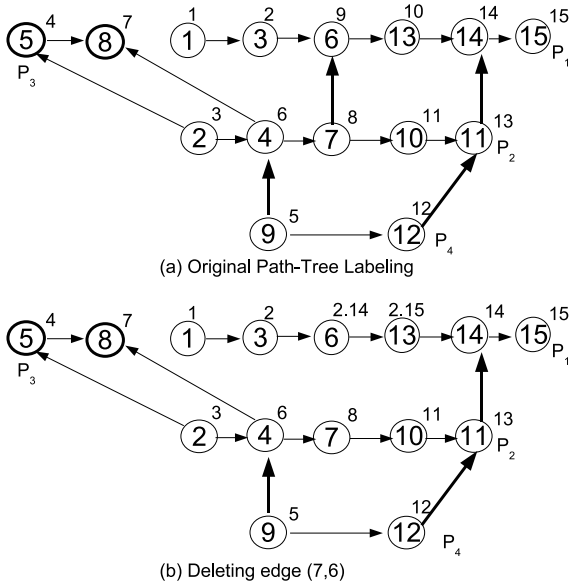


Fig. 38. An example of deleting a path-tree edge

The proof for Lemma 9 is basically a literal repeat of the proof of Lemma 8 and we omit it here.

Deletion of a non-path-tree edge

Deletion of a non-path-tree edge is easy to handle and there's no change on path-tree cover and labeling. Assume the edge to be deleted is uv. if u can reach v through the path-tree $G[T]$, then no update is necessary. Otherwise, we need to recompute the transitive closure as we did previously for deleting a path-tree edge.

Finally, it's necessary to mention that the incremental update algorithms in this section do not preserve optimality of path-tree cover. After sufficient updates, it is worthwhile to rebuild a new path-tree cover from the scratch, and the amortized cost would be acceptable.

D.3. Chain-Tree Updates

The update for a chain-tree can be done in two steps:

1. Determine if adding (or deleting) an edge (v_i, v_j) will result in an actual change of reachability from v_i to v_j.
2. If the above answer is no, do nothing; else, follow the path-tree update procedure as discussed previously, by viewing the chain-tree for G as a path-tree for G' (recall Section 4).

It only takes O(1) time to determine if adding an edge (v_i, v_j) will result in an actual change of reachability from v_i to v_j, if the transitive closure of G is available; otherwise, it takes $O(m+n)$ time to get an answer by Depth-First Search or Breadth-First Search.

Similarly, by Depth-First Search or Breadth-First Search, it takes $O(m + n)$ time to determine if deleting an edge (v_i, v_j) will result in an actual change of reachability from v_i to v_j in G.